Hibernate In Action

中文版

第一章 理解对象-关系持续性

我们工作的每个软件项目工程中，管理持续性数据的方法已经成为一项关键的设计决定。对于Java应用，持续性数据并不是一个新的或不寻常的需求，你也许曾经期望能够在许多相似的，已被很好构建的持续性解决方案中简单地进行选择。考虑一下Web应用框架(Jakarta Struts 对 WebWork)，GUI组件框架（Swing 对 SWT），或模版工具（JSP 对 Velocity）。每一种相互竞争的解决方案都有其优缺点，但它们至少都共享了相同的范围与总体的方法。不幸的是，这还不是持续性技术的情形，对持续性技术相同的问题有许多不同的混乱的解决方案。

在过去的几年里，持续性已经成为Java社区里一个争论的热点话题。对这个问题的范围许多开发者的意见甚至还不一致。持续性还是一个问题吗？它早已被关系技术与其扩展例如存储过程解决了。或者它是一个更一般的问题，必须使用特殊的Java组件模型例如EJB实体Bean来处理？甚至SQL和JDBC中最基本的CRUD（create, read, update, delete）操作也需要进行手工编码，还是让这些工作自动化？如果每一种数据库管理系统都有它自己的方言，我们如何达到可移植性？我们应该完全放弃SQL并采用一种新的数据库技术，例如面向对象数据库系统吗？争论仍在继续，但是最近一种称作对象-关系映射（ORM）的解决方案逐渐地被接受。Hibernate就是这样一种开源的ORM实现。

Hibernate是一个雄心勃勃的项目，它的目标是成为Java中管理持续性数据问题的一种完整的解决方案。它协调应用与关系数据库的交互，让开发者解放出来专注于手中的业务问题。Hibernate是一种非强迫性的解决方案。我们的意思是指在写业务逻辑与持续性类时，你不会被要求遵循许多Hibernate特定的规则和设计模式。这样，Hibernate就可以与大多数新的和现有的应用平稳地集成，而不需要对应用的其余部分作破坏性的改动。

本书是关于Hibernate的。我们包含了基本与高级的特征，并且描述了许多使用Hibernate开发新应用时的推荐方式。通常这些推荐并不特定于Hibernate——有时它们可能是我们关于使用持续性数据工作时处理事情的最佳方式的一些想法，只不过在Hibernate的环境中进行了介绍。然而，在我们可以开始使用Hibernate之前，你需要理解对象持续性和对象-关系映射的核心问题。本章解释了为什么像Hibernate这样的工具是必需的。

首先，我们定义了在面向对象的应用环境中持续性数据的管理，并且讨论了SQL，JDBC和Java的关系，Hibernate就是在这些基础的技术与标准之上构建的。然后我们讨论了所谓的对象-关系范例不匹配的问题和使用关系数据库进行面向对象的软件开发中所遇到的一些一般性的问题。随着这个问题列表的增长，我们需要一些工具与模式来最小化我们用在与持续性有关的代码上的时间就变得很明显了。在我们查看了可选的工具和持续性机制后，你会发现ORM在许多情况下可能是最好的解决方案。我们关于ORM的优缺点的讨论给了你一个完整的背景，在你为自己的项目选择持续性解决方案时可以作出最好的决定。

最好的学习方式并不必须是线性的。我们猜想你可能想立刻尝试一下Hibernate。如果这的确是你喜欢的方式，请跳到第2章第2.1节“开始”，在那儿我们进入并开始编写一个（小的）Hibernate应用。不读这一章你也可能理解第2章，但我们还是推荐你在循环通读本书的某一时刻再回到这一章。那样，你就可以准备好并且具有了学习其余资料所需的所有的背景概念。

1．1 什么是持续性？

几乎所有的应用都需要持续性数据。持续性在应用开发中是一个基本的概念。如果当主机停电时一个信息系统没有保存用户输入的数据，这样的系统几乎没有实际的用途。当我们讨论Java中的持续性时，我们通常是指使用SQL存储在关系数据库中的数据。我们从简单地查

看一下这项技术和我们如何在Java中使用它开始。具有了这些知识之后，我们继续关于持续性的讨论以及如何在面向对象的应用中实现它。

1．1．1 关系数据库

你，像许多其他的开发者，很可能在使用一个关系数据库进行工作。实际上，我们中的大多数每天都在使用关系数据库。关系技术是一个已知数。仅此一点就是许多组织选择它的一个充分的理由。但仅仅这样说就有点欠考虑了。关系数据库如此不易改变不是因为偶然的原因而是因为它们那令人难以置信的灵活与健壮的数据管理方法。

关系数据库管理系统既不特定于Java，也不特定于特殊的应用。关系技术提供了一种在不同的应用或者相同应用的不同技术（例如事务工具与报表工具）之间共享数据的方式。关系技术是多种不同的系统与技术平台之间的一个共同特征。因此，关系数据模型通常是业务实体共同的企业级表示。

关系数据库管理系统具有基于SQL的应用编程接口，因此我们称今天的关系数据库产品为SQL数据库管理系统，或者当我们讨论特定系统时，称之为SQL数据库。

1．1．2 理解SQL

为了有效地使用Hibernate，对关系模型和SQL扎实的理解是前提条件。你需要用你的SQL知识来调节你的Hibernate应用的性能。Hibernate会自动化许多重复的编码任务，如果你想利用现代SQL数据库的全部能力,你的持续性技术的知识必须扩充至超过Hibernate本身。记住基本目标是健壮地有效地管理持续性数据。

让我们回顾一些本书中用到的SQL术语。你使用SQL作为数据定义语言（DDL）通过CREATE与ALTER命令来创建数据库模式。创建了表（索引，序列等等）之后，你又将SQL作为数据处

理语言（DML）来使用。使用DML，你执行SQL操作来处理与取出数据。处理操作包括插入，更新和删除。你使用约束，投射，和连接操作（包括笛卡尔积）执行查询来取出数据。为了有效地生成报表，你以任意的方式使用SQL来分组，排序和合计数据。你甚至可以相互嵌套SQL命令，这项技术被称作子查询。你也许已经使用了多年的SQL并且非常熟悉用它编写的基本操作和命令。尽管如此，我们的经验还是告诉我们SQL有时难于记忆并且有些术语有不同的用法。为了理解本书，我们不得不使用相同的术语与概念；因此，如果我们提到的任何术语是新出现的或者是不清楚的，我们建议你看一下附录A。

为了进行健全的Java数据库应用开发，SQL知识是强制的。如果你需要更多的资料，找一本Dan Tow的优秀著作《SQL Tuning》。同时读一下《An Introduction to Database Systems》，学习一些（关系）数据库系统的理论，概念和思想。关系数据库是ORM的一部分，当然另一部分由你的Java应用中的对象构成，它们需要使用SQL被持续化到数据库中。

1．1．3 在Java中使用SQL

当你在Java应用中使用SQL工作时，Java代码通过Java数据库连接（JDBC）执行SQL命令来操作数据库。SQL可能被手工编码并嵌入到Java代码中，或者可能被匆忙地通过Java代码生成。你使用JDBC API将变量绑定到查询参数上，开始执行查询，在查询结果表上滚动，从结果集中取出值，等等。这些都是底层的数据访问任务，作为应用开发者，我们对需要这些数据访问的业务问题更加感兴趣。需要我们自己来关心这些单调的机械的细节，好像并不是确定无疑的。

我们真正想做的是能够编写保存和取出复杂对象（我们的类实例）的代码—从数据库中取出或者保存到数据库中，尽量为我们减少这些底层的苦差事。

因为这些数据访问任务通常都是非常单调的，我们不禁要问：关系数据模型特别是SQL是

面向对象的应用中持续性问题的正确选择吗？我们可以立即回答这个问题：是的！有许多原因使SQL数据库支配了计算行业。关系数据库管理系统是唯一被证明了的数据管理技术并且几乎在任何Java项目中都是一项需求。

然而，在过去的15年里，开发者一直在讨论范例不匹配的问题。这种不匹配解释了为什么每个企业项目都需要在持续性相关的问题上付出如此巨大的努力。这里所说的范例是指对象模型和关系模型，或者可能是面向对象编程与SQL。让我们通过询问在面向对象的应用开发环境中，持续性究竟意味着什么，来开始我们对不匹配问题的探究。首先，我们将本章开始部分声明的对持续性过分简单的定义扩展到一个较宽的范围，更成熟的理解包括维护与使用持续性数据。

1．1．4 面向对象应用中的持续性

在面向对象的应用中，持续性允许一个对象的寿命可以超过创建它的程序。这个对象的状态可能被存储到磁盘上，并且在将来的某一时刻相同状态的对象可以被重新创建。

这样的应用不仅仅限于简单的对象——关联对象的完整图形也可以被持续化并且以后可以在新的进程里被重新创建。大多数对象并不是持续性的；暂态对象只有有限的寿命，被实例化它的进程的寿命所限定。几乎所有的Java应用都在混合使用持续与暂态对象；因此，我们需要一个子系统来管理我们的持续性数据。

现代的关系数据库为持续性数据提供了一种结构化的表示方法，允许排序，检索和合计数据。数据库管理系统负责管理并发性和数据的完整性；它们负责在多个用户和多个应用之间共享数据。数据库管理系统也提供了数据级别的安全性。当我们在本书中讨论持续性时，我们考虑以下这些事情：

- 存储，组织与恢复结构化数据

- 并发性与数据完整性
- 数据共享

特别地，我们将在使用域模型的面向对象的应用环境中考虑这些问题。

使用域模型的应用并不直接使用业务实体的扁平表示进行工作，这些应用有它们自己的面向对象的业务实体模型。如果数据库中有“项目”与“竞价”表，则在这些Java应用中会定义“项目”与“竞价”类。

然后，业务逻辑并不直接在SQL结果集的行与列上进行工作，而是与面向对象的域模型进行交互，域模型在运行时表现为一个关联对象的交互图。业务逻辑从不在数据库中（作为SQL存储过程）执行，而是被实现在Java程序中。这就允许业务逻辑使用成熟的面向对象的概念，例如继承与多态。我们可以使用众所周知的设计模式例如“策略”,“中介者”和“组合” ，所有这些模式都依赖于多态方法调用。现在给你一个警告：并不是所有的Java应用都是按照这种方式设计的，并且也不打算是。对于简单的应用不使用域模型可能会更好。SQL和JDBC API可以完美地处理纯扁平数据，并且现在新的JDBC结果集已经使CRUD操作变得更简单了。使用持续性数据的扁平表示进行工作可能更直接并且更容易理解。

然而，对于含有复杂业务逻辑的应用，域模型有助于有效地提高代码的可重用性和可维护性。在本书中我们集中在使用域模型的应用上，因为通常Hibernate和ORM总是与这种类型的应用有关。

如果我们再次考虑SQL和关系数据库，我们最终会发现这两种范例的不匹配之处。

SQL操作例如投射与连接经常会导致对结果数据的扁平表示。这与在Java应用中用来执行业务逻辑的关联对象的表示完全不同！这是根本不同的模型，而不仅仅是相同模型的不同的显示方式。

带着这些认识，我们能够开始看一下这些问题——许多已经很好理解的与没有很好理解的——必须被合并使用这两种数据表示的应用所解决——一个面向对象的域模型与一个持续性的关系模型。让我们仔细地看一下。

1．2 范例的不匹配

范例不匹配的问题可以被分解成几部分，我们将依次进行分析。让我们通过一个独立于问题的简单例子开始我们的研究。随后当我们构建它时，你就会看到不匹配性问题的出现。

假设你需要设计与实现一个在线电子商务应用。在这个应用中你需要一个类来表示系统用户的信息，另一个类用来表示用户账单的详细资料（参见图**1.1**）。

Figure 1.1 A simple UML class diagram of the user and billing details entities

（图**1.1**）

看一下这个范例，你可以看到一个用户可以有多份账单资料，你可以在两个方向上对这两个类之间的关系进行导航。一开始，表示这些实体的类可能会非常简单：

```
public class User {
    private String userName;
    private String name;
    private String address;
    private Set billingDetails;
    ...
}

public class BillingDetails {
    private String accountNumber;
```

```
    private String accountName;
    private String accountType;
    private User user;
    ...
}
```

注意，我们仅对与持续性有关的实体状态感兴趣，因此我们省略了属性存取方法与业务方法（例如**getUserName()**或**billAuction()**）的实现。非常容易给这个例子提出一个好的**SQL**表设计：

```
create table USER (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    NAME VARCHAR(50) NOT NULL,
    ADDRESS VARCHAR(100)
)

create table BILLING_DETAILS (
    ACCOUNT_NUMBER VARCHAR(10) NOT NULL PRIMARY Key,
    ACCOUNT_NAME VARCHAR(50) NOT NULL,
    ACCOUNT_TYPE VARCHAR(2) NOT NULL,
    USERNAME VARCHAR(15) FOREIGN KEY REFERENCES USER
)
```

这两个实体之间的关系通过外键来表示，表**BILLING_DETAILS**中的**USERNAME**就是外键。对于这个简单的对象模型，对象**-**关系的不匹配几乎不明显；你可以直接编写**JDBC**代码来插入，更新和删除用户与账单资料的信息。

现在，让我们看一下当我们考虑一个更现实的问题时会发生什么。当我们在我们的应用中加入更多的实体和实体关系时，范例不匹配的问题会变得更明显。

在我们当前的实现中最明显的问题是我们将地址设计成了一个简单的字符串值。在大多

数系统里，并不需要分别保存街道、城市、州、国家和邮政编码等信息。当然，我们可以直接在用户类里增加这些属性，但是因为系统中其它的类也非常可能含有地址信息，创建一个地址类可能更有意义。更新后的对象参见图1.2。

Figure 1.2 The User has an Address.

（图1.2）

我们同样需要增加一个地址表吗？不需要。通常只需将地址信息作为单独的列保存到用户表里。这样的设计可能执行起来效率更高，因为在一个单独的查询里我们不需要通过表连接来取得用户和地址。最好的解决方案可能是创建一个用户定义的SQL数据类型来表示地址，并且在用户表里使用这种类型的一个单独的列而不是许多新列。

基本上，我们有增加几列或者使用单独的一列（以一种新的SQL数据类型）这两种选择。这明显是一个粒度的问题。

1．2．1 粒度的问题

粒度是指你使用的对象的相对大小。当我们讨论Java对象和数据库表时，粒度问题意味着持续性对象可以以不同的粒度来使用表和列，而表和列本身都有粒度上固有的限制。

让我们回到我们的例子。增加一种新的数据类型，将Java地址对象在我们的数据库中保存为单独的一列，听起来好像是最好的方法。毕竟，Java中的一个新的地址类与SQL数据类型中的一个新的地址类型可以保证互用性。然而，如果你检查现在的数据库管理系统对用户定义列类型（UDT）的支持，你将会发现各种各样的问题。

对传统SQL而言，UDT支持是许多所谓的对象-关系扩展中的一种。不幸的是，UDT支持在

大多数SQL数据库管理系统中都是有些晦涩的特征，当然在不同的系统之间也不具有可移植性。SQL标准支持用户定义的数据类型，但是少的可怜。因为这个原因或者（任何）其它原因，此时在本项工作中使用UDT还不是一项共通的实践——并且你不可能遇到一个大量使用了UDT的遗留系统。因此我们不能将我们新的地址类的对象作为一个等价的用户定义的SQL数据类型保存到一个独立的新列中。对这个问题我们的解决办法是使用多个列，将它们定义成厂方定义的SQL类型（例如布尔，数值与字符串数据类型）。同时考虑到表的粒度，用户表通常如下定义：

```
create table USER (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    NAME VARCHAR(50) NOT NULL,
    ADDRESS_STREET VARCHAR(50),
    ADDRESS_CITY VARCHAR(15),
    ADDRESS_STATE VARCHAR(15),
    ADDRESS_ZIPCODE VARCHAR(5),
    ADDRESS_COUNTRY VARCHAR(15)
)
```

这导致了下面的发现：我们的域对象模型中的类按照不同的粒度级别排列起来——从象用户这样的粗粒度的实体类到一个象地址这样的细粒度类，直到一个象邮政编码这样的简单的字符串属性值。

相比之下，在数据库的级别中只有两种粒度的级别是可见的：表例如用户和数量列例如邮政编码。这明显不如我们的Java类型系统灵活。许多简单的持续性机制未能识别这种不匹配性因此不再限制对象模型上的更不灵活的表示。我们曾经无数次地看到在用户类中包含一个邮政编码的属性。

可以证明粒度问题并不是特别难以解决。的确，我们甚至可能都不列出它，之所以列出

是因为在许多现有系统中它都非常明显这个事实。我们将在第3章，第3.5节“细粒度的对象模型”中描述这个问题的解决方案。

当我们考虑使用了继承的域对象模型时，一个更困难并且更有趣的问题就会出现。继承是面向对象设计的一个特征，我们可以用它以一种更新更有趣的方式给我们的电子商务应用的用户开账单。

1．2．2 子类型的问题

在Java中，我们使用超类与子类实现继承。为了介绍这为什么会出现不匹配性的问题，让我们继续构建我们的例子。扩展一下我们的电子商务应用以使我们现在不仅能接受银行账户账单，还能接受信用卡与借记卡。因此我们有几种方法可以给一个用户账户开账单。在我们的对象模型中反映这个改变的最普通的方式是继承账单详细资料类。

我们可能有一个抽象的账单详细资料的超类和几个具体的子类：信用卡，直接借记，支票等等。这些子类中的每一个都会定义一些稍微不同的数据（与处理这些数据的完全不同的功能）。图1.3的UML类图介绍了这个对象模型。

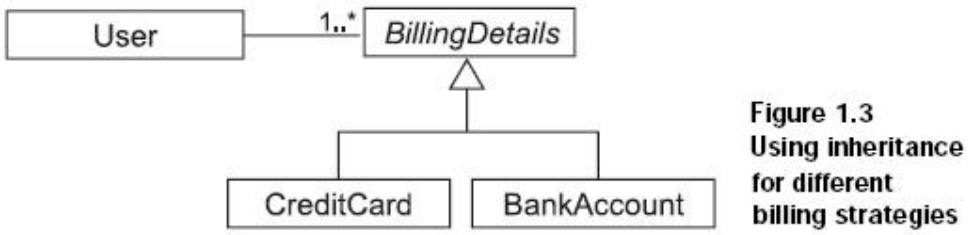

Figure 1.3
Using inheritance for different billing strategies

（图1.3）

我们立刻就会注意到SQL没有提供对继承的直接支持。我们不能够通过编写“CREATE TABLE CREDIT_CARD_DETAILS EXTENDS BILLING_DETAILS (...)”将表CREDIT_CARD_DETAILS声明为表BILLING_DETAILS的一个子类型。

在第3章第3.6节“映射类继承”里，我们讨论了象Hibernate这样的对象-关系映射解决方案如何解决将类层次持续化到数据库表中的问题。在社区里这个问题已经被很好地理解了，并且大多数的解决方案支持近似相同的功能。但是对于持续性问题我们还没有完全结束——只要我们将继承引入到对象模型中，就会出现多态的可能性。

用户类与账单详细资料超类有一个关联，这是一个多态性的关联。运行时，一个用户对象可能与账单详细资料的任何子类的实例相关联。相似地，我们希望能够针对账单详细资料类编写查询并且让它返回子类的实例。这个特征被称作多态性查询。

因为SQL数据库不支持继承的概念，它们没有明显的表示多态性关联的方式也不会太令人惊讶。一个标准的外键约束精确地引用一个表；定义一个引用多个表的外键并不是那么简单。我们可能以Java（与其它面向对象语言）的输入不如SQL那么严格来解释这个问题。幸运地是，我们将在第3章介绍两种继承映射的解决方案，它们被设计为用来解决多态性关联的表示和有效地执行多态性查询。

因此，子类型不匹配是指Java模型中的继承结构必须被持续化到SQL数据库中，而SQL数据库并没有提供一个支持继承的策略。不匹配问题的另一方面是对象的同一性问题。你可能已经注意到了我们将用户表中的用户名定义成了主键。那是一个好的选择吗？与你下面将要看到的一样，那的确不是。

1．2．3 同一性的问题

虽然对象同一性的问题一开始可能并不明显，但在我们不断增长与扩展的电子商务系统的例子中我们会经常遇到它。这个问题在我们考虑两个对象（例如，两个用户）并且检查它们是否是同一个时就会看到。有三种方法可以解决这个问题，两种使用Java，一种使用我们的SQL数据库。与你预期的一样，想让它们协同工作需要提供一些帮助。

Java对象定义了两种不同的相等性的概念：

- 对象同一性（粗略的等同于内存位置的相等，使用a==b检查）
- 通过equals()方法的实现来决定的相等性（也被称作值相等）

另一方面，数据库行的同一性使用主键值表示。象你将在第3.4节“理解对象同一性”中看到的一样，主键既不必然地等同于“equals()”也不等同于“==”。它通常是指几个对象（不相同的）同时表示了数据库中相同的行。而且，为一个持续类正确地实现equals()方法包含许多微妙的难点。

让我们通过一个例子讨论另一个关于数据库同一性的问题。在我们用户表的定义中，我们已经使用了用户名作为主键。不幸的是，这个决定使改变一个用户名变得很困难：我们不仅需要更新用户表中的用户名列，而且也要更新账单详细资料表中的外键列。因此，在本书后面的部分，我们推荐你无论什么情况如果可能的话都可以使用代替键。代替键是指对用户没有意义的主键列。例如，我们可以象这样改变我们的表定义：

```
create table USER (
    USER_ID BIGINT NOT NULL PRIMARY KEY,
    USERNAME VARCHAR(15) NOT NULL UNIQUE,
    NAME VARCHAR(50) NOT NULL,
    ...
)

create table BILLING_DETAILS (
    BILLING_DETAILS_ID BIGINT NOT NULL PRIMARY KEY,
    ACCOUNT_NUMBER VARCHAR(10) NOT NULL UNIQUE,
    ACCOUNT_NAME VARCHAR(50) NOT NULL,
    ACCOUNT_TYPE VARCHAR(2) NOT NULL,
    USER_ID BIGINT FOREIGN KEY REFERENCES USER
)
```

列USER_ID和BILLING_DETAILS_ID包含系统生成的值。引入这些列纯粹是为了关系数据模型的好处。它们如何（即使从不）在对象模型中表示？我们将在第3.4节讨论这个问题并且找到一个对象-关系映射的解决方案。

在持续性环境中，同一性与系统如何处理缓存和事务密切相关。不同的持续性解决方案选择不同的策略，这已经成为混乱的一方面。我们包含了所有这些有趣的主题——并显示了他们是如何相关的——在第5章。

我们已经设计和实现的电子商务应用的构架很好地达到了我们的目的。我们识别出了映射粒度，子类型和对象同一性这些不匹配性的问题。我们几乎准备好了转移到应用的其它部分。但是首先，我们需要讨论一下关联这个重要的概念——也就是说，我们的类之间的关系是如何被映射和处理的。数据库中的外键就是我们所需要的全部吗？

1．2．4 与关联有关的问题

在我们的对象模型中，关联描述了实体间的关系。你记得用户，地址和账单详细资料类都是相互关联的。与地址不同，账单详细资料依赖于自身。账单详细资料对象保存到它们自己的表中。在任何对象持续性解决方案中，实体关联的映射与管理都是中心概念。

面向对象的语言使用对象引用和对象引用的集合表示关联。在关系世界里，关联被表示为外键列，外键是几个表的键值的拷贝。这两种表示之间有些微妙的不同。

对象引用具有固有的方向性，关联是指从一个对象到另一个对象的引用。如果对象间的关联可以在两个方向上进行导航，你需要定义两次关联，在每个关联类上分别定义一次。这你早已在我们的对象模型类中见过了：

```
public class User {
    private Set billingDetails;
    ...
}

public class BillingDetails {
    private User user;
    ...
}
```

另一方面，外键并不通过固有的方向性进行关联。事实上，对关系数据模型来说导航没有任何意义，因为你可以通过表连接和投射创建任意的数据关联。

实际上，只看Java类不可能确定单向关联的多重度。Java关联可以具有多对多的多重度。例如，我们的对象模型可能有看起来像这样的：

```
public class User {
    private Set billingDetails;
    ...
}

public class BillingDetails {
    private Set users;
    ...
}
```

另一方面，表关联总是一对多或是一对一的。通过查看外键的定义你可以立刻知道多重度。下面是一个一对多的关联（或者，从另一个方向看，是多对一的）：

```
    USER_ID BIGINT FOREIGN KEY REFERENCES USER
```

这些是一对一的关联：

```
    USER_ID BIGINT UNIQUE FOREIGN KEY REFERENCES USER
    BILLING_DETAILS_ID BIGINT PRIMARY KEY FOREIGN KEY REFERENCES USER
```

如果你希望在关系数据库中表示一个多对多的关联，你必须引入一个新表，称为连接表。这个表不会在对象模型中的任何地方出现。对我们的例子来说，如果我们认为用户和用户的帐单信息之间为多对多的关系，连接表可以如下定义：

```
CREATE TABLE USER_BILLING_DETAILS (
    USER_ID BIGINT FOREIGN KEY REFERENCES USER,
    BILLING_DETAILS_ID BIGINT FOREIGN KEY REFERENCES BILLING_DETAILS
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

我们将会在第3和第6章非常详细地讨论关联映射。

目前为止，我们讨论的主要是结构上的问题。我们考虑系统的一个纯静态的视图就可以看到这些问题。或许在对象持续性中最困难的问题是动态的。它涉及到关联，在第1.1.4节“面向对象应用中的持续性”中，当我们描述对象图导航与表连接的区别时，就已经暗示过了。让我们更彻底地探讨这个重要的不匹配性问题。

1．2．5 对象图导航的问题

你在Java中访问对象的方式与在关系数据库中有根本的不同。在Java中，访问用户的账单信息时，你调用`aUser.getBillingDetails().getAccountNumber()`。这是最自然的面向对象数据的访问方式，通常被形容为遍历对象图。根据实例间的关联，你从一个对象导航到另一个对象。不幸地是，这不是从SQL数据库中取出数据的有效方式。

为了提高数据访问代码的性能，唯一重要的事是最小化数据库请求的次数。最明显的方式是最小化SQL查询的数量（其它方式包括使用存储过程或者JDBC批处理API）。

因此，使用SQL有效地访问关系数据通常需要在有关的表之间使用连接。在连接中包含的表的数量决定了你可以导航的对象图的深度。例如，如果我们需要取出用户，而对用户帐单的详细资料不感兴趣，我们使用这个简单查询：

```
select * from USER u where u.USER_ID = 123
```

另一方面，如果我们需要取出相同的用户并且随后访问每一个关联的账单详细资料的实例，我们使用一个不同的查询：

```
select *
from USER u
left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
where u.USER_ID = 123
```

像你看到的一样，当我们取出最初的用户时，在我们开始导航对象图之前，我们需要知道我们计划访问对象图的哪一部分。

另一方面，仅当第一次访问一个对象时，所有对象持续性解决方案才提供取出关联对象数据的功能。然而，在关系数据库环境中，这种逐步的数据访问风格基本上效率是比较低的，因为，它需要在对象图的每个节点上执行一条选择语句。这就是所谓的n+1次选择的问题。

我们在Java与关系数据库中访问对象方式的不匹配，可能是Java应用中性能问题的一个最普遍的根源。虽然我们有无数的书籍与杂志论文的保佑，建议我们使用StringBuffer进行字符串连接，但好像不可能找到任何关于避免n+1次选择问题的策略。非常幸运，Hibernate提供了有效地从数据库中取出对象图的成熟的特征，使应用可以透明地访问这些图。我们将在第4和第7章讨论这些特征。

现在我们已经有了一个非常详细的对象-关系不匹配问题的列表，就像你根据经验得知的

那样，为这些问题寻找解决方案的代价可能是非常高的。这个代价经常被低估，我们认为这是大多数软件项目失败的主要原因。

1．2．6 不匹配的代价

对不匹配问题列表的全面的解决方案可能需要花费大量的时间和努力。根据我们的经验，在Java应用中编写的可能达到30%的代码，主要的目的是为了处理单调的SQL/JDBC和对象-关系范例不匹配的手工连接。不管所有这些努力，最终的结果可能仍然感觉并不完全正确。我们曾经见过一些因为数据库抽象层的复杂性和僵硬性而几乎要垮掉的项目。

一个主要的代价在模型化方面。关系与对象模型必须包含相同的业务实体。与一个有经验的关系数据建模者相比，一个面向对象的纯粹主义者可能以一种非常不同的方式模型化这些实体。对这个问题，通常的解决方案是扭曲对象模型直到它与下层的关系技术匹配为止。

这可能会成功，但代价是损失了许多对象方向的优点。记住，关系模型有关系理论所支持。对象方向上没有如此严格的数学定义和理论工作的支柱。因此，我们不能期望使用数学来解释我们应该如何融合这两种范例——没有优雅的变化等待去发现（放弃Java和SQL从头开始并不是一种深思熟虑过的优雅的方式）。

域模型不匹配的问题不是僵硬性的唯一原因，它使生产性降低并导致了更高的成本。另一个原因是JDBC API本身。JDBC和SQL提供了一个面向语句（即命令）的方法从SQL数据库中来回移动数据。至少在三个时刻（Insert，Update，Select）必须指定一个结构化关系，这增加了设计和实现所需要的时间。每种SQL数据库的独特的方言并没有改善这种情况。

最近，考虑以体系结构或基于模式的模型作为对不匹配问题的部分的解决方案已经开始流行。因此，我们有了实体bean组件模型，数据访问对象（DAO）模式，和其它实现数据访问

的实践。这些方法将以前列出的大部分或全部的问题留给了应用开发者。为了转向对对象持续性的理解，我们需要讨论应用体系结构和在典型的应用设计中持续层的角色。

1．3 持续层与可选方案

在一个大、中型的应用中，根据关系来组织类通常很有意义。持续性就是一种关系。其它的关系包括表示、工作流和业务逻辑。也有所谓的“**cross-cutting**”关系，通常这可能由框架代码实现。典型的“**cross-cutting**”关系包括日志，验证和事务划分。

一个典型的面向对象的体系结构包含了表示这些关系的层。通常，当然也是一项最佳实践，是在一个分层的系统体系结构中将所有与持续性有关的类和组件分组到一个独立的持续层中。

在本节中，我们首先查看一下这种体系结构中的层和我们为什么要使用它们。然后，我们集中到我们最感兴趣的层——持续层——和一些可以实现它的方式。

1．3．1 分层的体系结构

分层的体系结构定义了实现不同关系的代码之间的接口，允许关系实现方式的改变不会对其它层的代码造成重大的破坏。分层也决定了其间出现的中间层的类型。规则如下：

- 层由上到下进行通信。每一层仅依赖于其直接的下层。
- 除了其直接下层，每一层都不知道任何其它层。

不同的应用以不同的概念分组，因此会定义不同的层。一个典型的，已被证明的，高层的应用体系结构使用三层，分别为表示层，业务逻辑层和持续层，参见图**1.4**。

让我们仔细地看一下图中的层和元素：

■ 表示层——最高层的用户接口逻辑。负责显示、页面控制和屏幕导航的代码构成了表示层。

■ 业务层——下一层的确切形式，不同的应用可能会有很大的差异。它通常是已协商一致的，然而，作为问题域的一部分业务层应该被用户所理解，它负责实现任何业务规则或系统需求。在许多系统里，这一层有它自己的业务域实体的内部表示。在其它的系统里，它重用了表示层定义的模型。我们将在第3章再次讨论这个问题。

■ 持续层——持续层是负责从一个或多个数据仓库中存入和取出数据的一组类和组件。它必须包含一个业务域实体模型（甚至只是一个元数据模型）。

■ 数据库——数据库位于Java应用之外。它是系统状态持续性的实际表示。如果使用了一个SQL数据库，它会包含关系模式或者可能有存储过程。

■ 帮助类/工具类——每个应用都会有一组基础的帮助类或工具类，在应用的每一层它们都会被使用（例如，用于异常处理的异常类）。这些基础元素并不构成一层，因此在分层的体系结构中，它们不用遵守中间层的依赖规则。

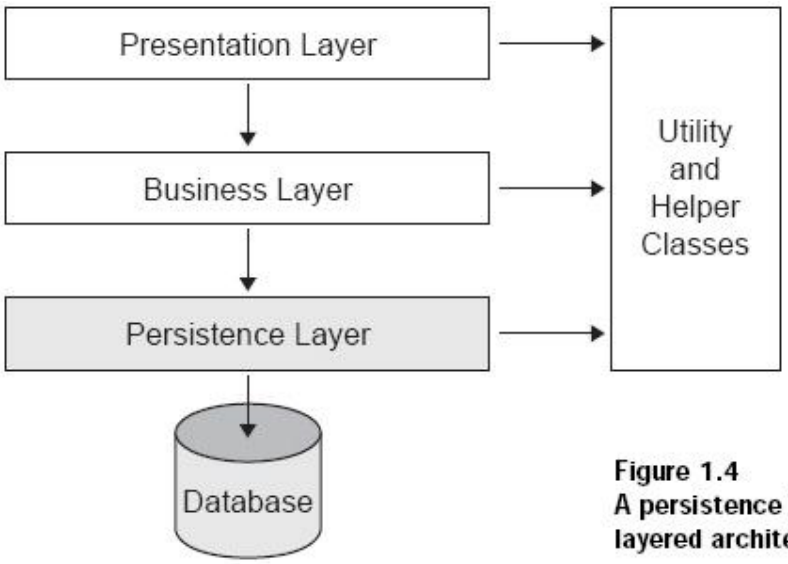

Figure 1.4
A persistence layer is the basis in a layered architecture.

（图1.4）

让我们简单地看一下Java应用实现持续层的不同方式。不必担心——我们将很快开始ORM和Hibernate。查看一下其它方法可以使我们学到很多东西。

1．3．2 使用SQL/JDBC手工编码持续层

实现Java持续性的最普遍的方式可能是应用程序员直接使用SQL和JDBC进行工作。毕竟，开发者熟悉关系数据库管理系统，理解SQL，也知道如何使用表和外键进行工作。此外，他们可以始终使用众所周知并广泛使用的DAO设计模式对业务逻辑隐藏复杂的JDBC代码和不可移植的SQL。

DAO是一种很好的模式——如此之好以至于我们甚至推荐与ORM一起使用它。然而，为域中的每个类手工编写持续性代码的工作是非常可观的，特别是需要支持多种SQL方言时。这项工作通常会消耗很大一部分的开发努力。此外，当需求改变时，一个手工编码的解决方案总是需要更多的注意和维护努力。

因此为什么不实现一个简单的ORM框架来满足你的项目的特定需求呢？这种努力的结果甚至可以在以后的项目中被重用。许多开发者已经采取了这种方法；在现今的产品系统中有许多自制的对象-关系持续层。然而，我们并不推荐这种做法。优秀的解决方案早已存在了，不仅有由商业厂商销售的工具（通常非常昂贵）而且也有使用自由许可证发布的开源项目。我们确信不论在业务或技术上，你都能够找到一种满足需求的解决方案。有可能这样的解决方案会比你在有限的时间里构建的做得更多，做得更好。

开发一个适度的包含全部特征的ORM可能需要花费很多人月。例如，Hibernate包含43,000行代码（其中许多比典型应用的代码要困难得多）和12，000行的单体测试代码。这可能远远多于你的应用。大量的细节很容易被忽略——象所有开发者从经验中得知的那样！即使一个现有的工具没有完全实现你的两三个更奇异的需求，也根本不值得自己创建。任何ORM都会处

理这些单调的普遍的情况——正是这些真正地破坏了生产性。你也可能需要针对特殊情况进行手工编码；很少应用主要由特殊情况构成。

不要被“现在尚未发明”的综合症所欺骗，就开始你自己的对象-关系映射的努力，而这只是为了避免第三方软件的学习曲线。即使如果你断定所有的ORM的材料都很狂热，并且你想以尽可能接近SQL数据库的方式工作，其它现有的持续性框架也都没有实现完全的ORM。例如，iBATIS数据库层是一个开源的持续层，它处理了许多更单调的JDBC代码但却让开发者手工编写SQL。

1．3．3 使用序列化

Java有一个内建的持续性机制：序列化提供了将对象图（应用状态）写到字节流中的能力，然后它可能被持续化到文件或数据库中。序列化也被Java的远程方法调用（RMI）使用来为复杂对象传递值语义。序列化的另一种用法是在机器集群中跨节点复制应用状态。

为什么不在持续层中使用序列化呢？很不幸，一个相互连接的对象图在序列化之后只能被当作一个整体访问，如果不反序列化整个流就不可能从流中取出任何数据。这样，结果字节流肯定会被认为不适合进行任意的检索或聚合。甚至不可能独立地访问或更新一个单独的对象或子图。为了支持高并发性，在进行系统设计时，除了在每个事务中装载与重写一次完整的对象图外没有其它的选择。

非常明显，因为当前特定的技术，序列化不适合于作为高并发性的Web和企业应用的持续性机制。在特定的环境中它被作为桌面应用的适当的持续性机制。

1．3．4 考虑EJB实体Bean

最近几年，企业JavaBeans（EJB）已经成为持续性数据的推荐方式。如果你一直在Java

企业应用领域工作，你可能已经使用过EJB特别是实体Bean了。如果没有，你也不用担心——实体Bean的流行度正在迅速地下降（尽管开发者大部分的关注还会集中在新的3.0规范上）。

实体Bean（在当前的EJB2.1规范中）非常有趣，因为与这里提到的其它解决方案相比，它们完全是通过委员会创建的。其它的解决方案（DAO模式，序列化和ORM）则是根据多年的经验提炼出来的；他们代表了经受了时间考验的方法。或许并不令人惊讶，EJB2.1实体Bean在实践中彻底地失败了。EJB规范的设计缺陷阻碍了Bean管理的持续性（BMP）实体Bean有效地执行。在EJB1.1许多明显的缺陷被纠正之后，一个边缘的稍微可接受的解决方案是容器管理的持续性（CMP）。

然而，CMP并不能表示一种对象-关系不匹配的解决方案。

这儿有六方面的原因：

■ CMP Bean与关系模型中的表是按照一对一的方式定义的。这样，它们的粒度过粗，不能够完全利用Java丰富的类型。在某种意义上，CMP强迫你的域模型成为最初通常的格式。

■ 另一方面，使用CMP Bean来实现EJB宣称的目标——可重用的软件组件，又显得粒度太细。一个可重用的组件必须是一个粒度非常粗的对象，面对数据库模式的一些小的改变它必须提供一个稳定的外部接口（是的，我们的确刚声明过CMP实体Bean的粒度既太粗又太细）。

■ 虽然EJB可以利用继承实现，但实体Bean并不支持多态的关联和查询——真正的ORM定义的特征。

■ 不管EJB规范所宣称的目标，实体Bean实际上是不可移植的。CMP引擎的性能因厂商而异，并且映射元数据也是高度特定于厂商的。许多项目选用Hibernate是因为Hibernate应用在不同的应用服务器之间具有更高的可移植性这个简单的原因。

■ 实体Bean不可序列化。我们发现当我们需要将数据传输到远程客户层时，我们必须定义额外的数据传输对象（DTO，也被称作值对象）。从客户端到远程实体Bean的实例使用细粒

度的方法调用是不可扩展的，DTO提供了一种远程数据访问的批处理方式。DTO模式导致了并行的类层次的增长，域模型中的每一个实体都要同时使用一个实体Bean和一个DTO来表示。

■ EJB是一种插入式模型；它要求一种不自然的Java风格并且在特定容器之外重用代码极其困难。对于测试驱动开发来说这是一个巨大的障碍。在需要批处理或其它离线功能的应用中，它甚至可能会引起问题。

我们不会花费更多的时间讨论EJB2.1实体Bean的优缺点。在查看了他们的持续性能力之后，我们得出了它们不适合完全的对象映射的结论。我们将拭目以待新的EJB3.0规范会有哪些提高。让我们转向另一种值得注意的对象持续性解决方案。

1．3．5 面向对象的数据库系统

因为我们使用Java对象工作，如果有一种方式根本不需要扭曲对象模型就能将这些对象存储到数据库中，那将是非常理想的。在90年代中期，新的面向对象的数据库系统引起了人们的注意。

与外部数据仓库相比，面向对象的数据库管理系统（OODBMS）更像是应用环境的扩展。通常OODBMS以一个多层实现作为主要特征，包含一个后端数据仓库，对象缓存，以及一个紧密结合的通过私有网络协议交互的客户端应用。

面向对象的数据库开发首先是宿主语言绑定的自上而下的定义，这些绑定为编程语言增加了持续性能力。因此，对象数据库提供了与面向对象应用环境无缝的集成。这不同于当今关系数据库使用的模型，它们通过中间语言（SQL）与数据库交互。

与ANSI SQL——关系数据库的标准查询接口类似，对象数据库产品也有标准。对象数据管理组（ODMG）规范定义了一组API，包括一种查询语言，一种元数据，以及C++、SmallTalk

和Java的宿主语言绑定。大多数面向对象的数据库系统对ODMG标准都提供了许多程度的支持，但据我们所知，还没有完全的实现。此外，在规范发布以后的很多年，甚至到了3.0版，还是感觉不太成熟，并且缺乏很多有用的特征，特别是基于Java环境的。ODMG也不再活跃。最近，Java数据对象（JDO）规范（发表于2002年4月）揭开了新的可能性。JDO由面向对象数据库团体的成员驱动，除了对现有的ODMG的支持之外，面向对象的数据库产品现在还经常将其作为主要的API采用。是否这些新的努力能够使面向对象数据库渗入到CAD/CAM（计算机辅助设计/建模）、科学计算以及其它市场环境中，还有待观察。

我们不会更进一步地查看为什么面向对象的数据库技术没有流行起来——我们只是简单地观察到对象数据库现在还没有被广泛地采用并且看来近期内也不太可能。根据当前行政上的实际情况（预定义的开发环境），我们确信绝大多数开发者还会有很多使用关系技术工作的机会。

1．3．6 其它选择

当然，也有其它类型的持续层。XML持续层是序列化模式的变种；这种方法通过允许工具可以很容易地访问数据结构，解决了一些字节流序列化的限制（但是它本身导致了对象/层次的不匹配）。此外，使用XML并没有其它的好处，因为它仅仅是另外一种文本文件格式。你也可以使用存储过程（甚至可以在Java里使用SQLJ编写）将这些问题移到数据库层。我们确信还有许多其它的例子，但最近没有哪一种可能会流行起来。

行政上的限制（在SQL数据库上长期的投资）与访问有价值的遗留数据的需求都要求一种不同的方法。对于我们的问题，ORM可能是最现实的解决方案。

1．4 对象-关系映射

至此，我们已经查看了对象持续性的可选技术，现在是介绍我们感觉最好的，并且是Hibernate使用的解决方案（ORM）的时候了。不管它很长的历史（第一篇研究论文发表于80年代后期），开发者对这个术语的使用也不相同。一些称它为“对象关系映射”，另一些则喜欢更简单的“对象映射”。我们统一使用术语“对象-关系映射”和它的首字母缩写ORM。中间的短线强调了当这两个领域相碰撞时出现的不匹配问题。

本节，我们首先查看一下什么是ORM。然后我们列举出一个好的ORM解决方案需要解决的问题。最后，我们讨论ORM提供的大致的好处与我们为什么推荐这种解决方案。

1．4．1 什么是ORM？

简单地说，对象-关系映射就是Java应用中的对象到关系数据库中的表的自动的（和透明的）持续化，使用元数据对对象与数据库间的映射进行了描述。本质上，ORM的工作是将数据从一种表示（双向）转换为另一种。

这意味着有一些性能损失。然而，如果ORM是作为中间件实现的，就会有许多机会可以进行优化而在手工编码的持续层中这些机会是不存在的。另外一项开销（在开发时）是对控制转换的元数据的准备与管理。而且，这个成本低于维护一个手工编码的解决方案所需的成本。相比之下，与ODMG兼容的对象数据库甚至需要大量类级别的元数据。

FAQ ORM不是一个Visio插件吗？首字母缩略语ORM也可以指对象角色建模，而且这个术语是在相关的对象-关系映射之前发明的。它描述了一种在数据库建模中使用的信息分析方法，并且主要由微软的Visio（一种图形化建模工具）支持。数据库专家使用它作为更流行的实体-关系建模的替代品或附属品。然而，如果你与Java开发者讨论ORM的话，通常是在对象-关系

映射的环境里。

ORM解决方案有以下四部分组成：

- 在持续类的对象上执行基本的CRUD操作的一组API。
- 用于指定查询的一种语言或一组API，这些查询会引用类和类属性。
- 用于指定映射元数据的工具。
- 实现ORM的一项技术，用来与事务对象交互以完成脏检查、懒关联存取和其它优化功能。

我们使用ORM这个术语包含所有可以根据元数据的描述自动生成SQL的持续层。我们不包含开发者通过编写SQL和使用JDBC手工解决对象-关系映射问题的持续层。使用ORM，应用与ORM API和根据下层SQL/JDBC抽象出来的域模型类进行交互。依赖于这些特征或特定的实现，ORM运行时也可能承担例如乐观锁、缓存等问题的职责，完全免去了应用对这些问题的关心。

让我们看一下可以实现ORM的各种不同的方式。Mark Fussel,一位ORM领域的研究者，定义了ORM品质的四个级别。

纯关系

整个应用，包括用户接口，都是围绕关系模型和基于SQL的关系操作进行设计的。如果低水平的代码重用是可以容忍的，不考虑它对大型系统的不足，这种方法对于简单的应用不失为一种优秀的解决方案。直接的SQL可以在每一方面进行微调，但是这些缺点例如缺乏可移植性和可维护性是非常重要的，特别是需要长期运行时。这种类型的应用经常大量地使用存储过程，将业务层的许多工作移动到了数据库中。

轻量对象映射

实体作为类来表示，而类又被手工地映射到关系表。手工编码的SQL/JDBC使用众所周知的设计模式对业务逻辑进行了隐藏。这种方法非常普遍并且对于那些只有少量实体的应用或者那些使用普通的元数据-驱动的数据模型的应用来说是很成功的。在这种类型的应用中存储

过程也可能有一席之地。

中等对象映射

这种应用是围绕对象模型设计的。SQL在编译时使用代码生成工具生成，或者在运行时由框架代码生成。对象间的关联由持续性机制支持，并且查询可能使用面向对象的表达式语言指定。对象被持续层缓存。很多的ORM产品和自制的持续层都至少支持这一级别的功能。它非常适合包含一些复杂事务的中等规模的应用，特别是当不同的数据库产品间的移植性非常重要时。这些应用通常不使用存储过程。

完全对象映射

完全的对象映射支持复杂的对象模型：组合、继承、多态和“可达的持续性”。持续层实现了透明的持续性；持续类不必继承任何特定的基类或实现任何特定的接口。高效的存取策略（懒惰和即时存取）和缓存策略都对应用透明地实现了。这一级别的功能很难由自制的持续层达到——它相当于数月或数年的开发时间。许多商业的和开源的Java ORM工具已经达到了这一级别的品质。这一级别满足了我们在本书中使用的ORM的定义。让我们看一下我们期望使用达到了完全对象映射级别的工具所能解决的一些问题。

1．4．2 一般的ORM问题

下面的问题列表，我们称之为对象-关系映射问题，是Java环境中完全的对象-关系映射工具解决的一些基本问题。特殊的ORM工具可能提供额外的功能（例如，积极缓存）。这是一个相当详细的概念问题的列表，而且是对象-关系映射特定的：

1 持续类像什么？它们是细粒度的JavaBean吗？或者它们是一些类似于EJB的组件模型的实例吗？持续性工具有多么透明？我们需要为业务领域的类采用一种编程模型或一些规范吗？

2 映射元数据是如何定义的？因为对象-关系转换完全由元数据控制，这些元数据的格式和定义是重要的核心问题。ORM工具应该提供一个图形化处理元数据的GUI吗？或者有定义元数据的更好的方法吗？

3 我们应该映射类的继承层次吗？这有几种标准策略。多态关联、抽象类和接口怎么映射呢？

4 对象同一性和相等性如何关联到数据库同一性（主键）？我们如何将特定类的实例映射到特定表的行。

5 在运行时持续性逻辑如何与业务域对象交互？这是一个普通的编程问题，有许多的解决方案包括源代码生成、运行时反射、运行时字节码生成和编译时字节码增强。这个问题的解决方案可能影响到你的构建过程（但宁可如此，你也不愿受到其它像用户那样的影响）。

6 持续性对象的生命周期是什么样的？有些对象的生命周期依赖于其它关联对象的生命周期吗？我们如何将一个对象的生命周期转化为数据库行的生命周期？

7 为排序、检索和合计提供了什么样的工具？应用可以在内存中处理其中的一些事情。但为了有效地使用关系技术有时需要通过数据库完成这些工作。

8 我们如何有效地取出关联数据？对关系数据的有效访问通常通过表连接实现。面向对象的应用通常通过导航对象图访问数据。可能的话，两种数据访问模式应该避免n+1次选择的问题，以及它的补充笛卡尔积的问题（在一次查询中取出过多的数据）。

另外，有两个问题对任何数据访问技术都是共通的。它们在ORM的设计和架构上强加了一些基本的限制。

■ 事务和并发性

■ 缓存管理（和并发性）

像你能够看到的一样，一个完全的对象映射工具需要处理一个相当长的问题列表。我们将在第3、4、5章讨论Hibernate管理这些问题的方式和一些数据访问的问题，稍后我们在本书中展开这个主题。

到此，你应该可以看到ORM的价值了。下一节，我们看一下当你使用ORM解决方案时，你将得到的一些其它好处。

1．4．3 为什么选择ORM？

ORM的实现是非常复杂的——可能不如应用服务器复杂，但要比像Struts或Tapestry这样的Web应用框架复杂得多。为什么我们要将另一个新的复杂的基础元素引入我们的系统呢？这样做值得吗？

对这些问题提供一个完全的回答将占用本书大部分的篇幅。为了避免你不耐烦，本章对大多数引人注目的好处提供了一个快速的概览。但首先，让我们快速地处理一些不是好处的问题。

ORM的一个假定的优点是对棘手的SQL保护开发者。这种观点认为面向对象的开发者不期望对SQL或关系数据库有很好的理解，而且不知为什么他们发现SQL非常令人讨厌。正好相反，我们认为Java开发者为了使用ORM工作，必须要充分熟悉——和感激——关系模型和SQL。ORM是开发者使用的一项高级技术，而且早以困难的方式实现过。为了有效地使用Hibernate，你必须能够查看和解释SQL语句的问题并能理解其性能含义。

让我们看一些ORM和Hibernate的好处。

生产性

与持续性有关的代码可能是Java应用中最乏味的代码。Hibernate去掉了很多让人心烦的工作（多于你的期望），让你可以集中到业务问题上。不论你喜欢哪种应用开发策略——自顶向下，从域模型开始；或者自底向上，从一个现有的数据库模式开始——使用Hibernate和适当的工具将会减少大量的开发时间。

可维护性

更少的代码行数（LOC）使系统更容易理解因为它们强调了业务逻辑而不是管道设备。更重要的，一个系统包含的代码越少则越容易重构。自动的对象-关系持续性充分地减少了LOC。当然，统计代码行数是度量应用复杂性的值得争议的方式。

然而，Hibernate应用更容易维护有其它方面的原因。在手工编码的持续性系统中，关系表示和对象模型之间存在一种不可避免的紧张。改变一个几乎总是包含改变其它的。并且一种表示设计经常需要妥协来适应其它的存在（实际上几乎总是发生的是域对象模型进行妥协）。ORM在这两种模型之间提供了一个缓冲，允许Java一方更优雅地进行面向对象的使用，并且每个模型都对其它模型的轻微改动进行了绝缘。

性能

一个共同的断言是手工编码的持续性与自动的持续性相比可能至少一样快，并且经常更快一些。在相同的意义上：汇编代码至少与Java代码一样快，或者手工编写的解析器至少与YACC或ANTLR生成的一样快，这的确是真的——换句话说，这有点离题了。这种断言未明确说明的含意是在实际的应用中手工编码的持续性至少应该完成得一样好。但这种含意只有在实现至少一样快的手工编码的持续性所需的努力与利用自动的解决方案包含的努力相似的情况下才是对的。实际令人感兴趣的问题是，当我们考虑时间和预算的限制时会发生什么？

对一项给定的持续性任务，可以进行许多优化。许多（例如查询提示）通过手工编码的

SQL/JDBC也很容易完成，然而，使用自动的ORM完成会更简单。在有时间限制的项目中，手工编码的持续层通常允许你利用一点时间做一些优化。Hibernate则允许你在全部的时间内做更多的优化。另外，自动的持续性给开发者提供了如此高的生产性因此你可以花更多的时间手工优化一些其余的瓶颈。

最后，ORM软件的实现人员可能有比你更多的时间来研究性能优化问题。你知道吗，例如，缓存PreparedStatement的实例对DB2的JDBC驱动导致了一个明显的性能增长但却破坏了InterBase的JDBC驱动？你了解吗，对某些数据库只更新一个表中被改变的字段可能会非常快但潜在地对其它的却很慢？在你手工编写的解决方案中，对这些不同策略之间的冲突进行试验是多么不容易呀？

厂商独立性

ORM抽象了你的应用使用下层SQL数据库和SQL方言的方式。如果工具支持许多不同的数据库（大部分如此），那么这会给你的应用带来一定程度的可移植性。你不必期望可以达到“一次编写，到处运行”，因为数据库的性能不同并且达到完全的可移植性需要牺牲更强大的平台的更多的力气。然而，使用ORM开发跨平台的应用通常更容易。即使你不需要跨平台操作，ORM依然可以帮你减小被厂商锁定的风险。另外，数据库独立性对这种开发情景也有帮助：开发者使用一个轻量级的本地数据库进行开发但实际产品需要配置在一个不同的数据库上。

1.5 总结

在本章中，我们讨论了对象持续性概念和ORM作为一项实现技术的重要性。对象持续性意味着对象个体可以比应用进程的寿命更长；它们可以被保存到数据仓库里并且以后可以被及时重建。当数据仓库是基于SQL的关系数据库管理系统时对象/关系不匹配的问题就会出现。例如，对象图不能被简单地保存到数据库表里；它必须被分解并被持续化到轻便的SQL数据类

型里。，ORM是这个问题的一个很好的解决方案，当我们考虑类型丰富的Java域模型时它特别有用。

域模型表示了在Java应用中使用的业务实体。在一个分层的系统体系结构中，域模型用来执行业务层中的业务逻辑（在Java中，而不是在数据库中）。为了装载与存储域模型中的持续性对象，业务层与其下面的持续层进行通信。ORM是持续层中管理持续性的中间件。

ORM并不是所有持续性任务的银弹；它的目标是减少开发者95%的对象持续性工作，例如使用多表连接编写复杂的SQL语句或将值从JDBC结果集中拷贝到对象或对象图中。一个包含全部特征的ORM中间件可能提供数据库间的可移植性，特定的优化技术，例如缓存和其它不容易在有限的时间里使用SQL和JDBC手工编码的可行的功能。

可能有一天会出现一种比ORM更好的解决方案。我们（和其他很多人）或许必须重新考虑我们知道的关于SQL，持续性API标准和应用集成的每一件事。现在的系统将会进化成纯关系数据库系统与面向对象的无缝的集成还仅仅只是推测。但是，我们不能等待，并且没有任何迹象表明这些问题会很快地改善（一个数十亿美元的产业不会是非常灵活的）。ORM是当前可用的最好的解决方案，它是每天都要面对对象-关系不匹配问题的开发者节省时间的一种方式。

Hibernate In Action

中文版

第二章 引入与集成Hibernate

理解一些Java应用对对象-关系映射的需求是对的，但你可能更急于看到活动的Hibernate。让我们从一个简单的例子开始，展示它的一些能力。

你可能知道，编程书籍通过一个“Hello World”例子开始是一项传统。在本章中，我们将遵循这个传统，使用一个相对简单的“Hello World”程序来介绍Hibernate。然而，简单地在控制台窗口上打印一条消息并不足以实际地展示Hibernate。相反，我们的程序会将新建的对象存储到数据库中，更新它们，并且执行查询从数据库中取出它们。

本章是以后各章的基础。除了这个规范的“Hello World”例子，我们还介绍了核心的Hibernate API，并说明了如何在各种不同的运行环境例如J2EE应用服务器和独立的应用中配置Hibernate。

2. 1 Hibernate中的“Hello World”

Hibernate应用定义了映射到数据库表的持续类。我们的“Hello World”例子由一个类和一个映射文件组成。让我们看一下一个简单的持续类是什么样子的，映射是如何指定的，和我们使用Hibernate持续类的实例可以做的一些其它的事情。

这个简单应用的目标是在数据库中存储消息并且取出它们进行显示。这个应用有一个简单的持续类“Message”，表示了这些用于打印的消息。我们的Message类如清单2.1所示。

```
package hello;
public class Message {
    // 标识符属性
    private Long id;
```

```
    // 消息文本
    private String text;
    // 另一个消息的引用
    private Message nextMessage;
    private Message() {}
    public Message(String text) {
        this.text = text;
    }
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }
    public Message getNextMessage() {
        return nextMessage;
    }
    public void setNextMessage(Message nextMessage) {
        this.nextMessage = nextMessage;
    }
}
```

（清单2.1）

我们的Message类有三个属性：标识符属性，消息文本和另一个消息的引用。标识符属性允许应用访问数据库识别——持续性对象的主键。如果两个Message实例具有相同的标识符值，则它们表示了数据库中相同的行。我们将标识符属性的类型定义成了Long，但这并不是必须的。实际上，像你将要看到的一样，Hibernate允许任意的标识符类型。

你可能已经注意到了Message类的所有属性都具有JavaBean风格的属性访问方法。这个类也包含一个没有参数的构造方法。在我们的例子中使用的持续类几乎总是与此有些相似。

Message类的实例可以由Hibernate进行管理（对其持续化），但这也不是必须的。因为Message对象没有实现任何Hibernate特定的类或接口，我们可以像任何其它Java类那样使用它：

```
Message message = new Message("Hello World");
System.out.println( message.getText() );
```

这段代码精确地完成了我们刚提到过的对“Hello World”应用的期望：将“Hello World”打印到控制台上。这有点像我们在自作聪明；实际上，我们展示了Hibernate区别于其它一些持续性解决方案例如EJB实体Bean的一项重要特征。从根本上讲我们的持续类可以在任意的执行环境中使用——不需要专门的容器。当然，你到这里来是想看一下Hibernate，因此让我们将一个新的Message保存到数据库里：

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
session.save(message);
tx.commit();
session.close();
```

这段代码调用了Hibernate的Session和Transaction接口（很快我们就会开始介绍getSessionFactory()方法）。它与下面这条SQL语句的执行结果相似：

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (1, 'Hello World', null)
```

继续——Message_ID字段的初始值非常奇怪。我们没有在任何地方设置Message类的id属

性，因此我们可能认为它是null，对吗？实际上，id属性是比较特别的：它是一个标识符属性——它持有一个生成的唯一值（稍候，我们将讨论这个值是如何生成的）。当调用save方法时，这个值通过Hibernate赋给了Message类的实例。

对这个例子来说，我们假定MESSAGES表早已存在了。在第9章，我们将说明只使用映射文件里的信息，如何让Hibernate自动创建你的应用所需的表（不用你手工编写更多的SQL）。当然，我们希望我们的“Hello World”程序能将消息打印到控制台上。现在我们的数据库里已经有了一条消息，我们准备进行展示了。下一个例子按字母顺序从数据库中取出所有的消息，并打印它们：

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages =
        newSession.find("from Message as m order by m.text asc");
System.out.println( messages.size() + " message(s) found:" );
for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
    Message message = (Message) iter.next();
    System.out.println( message.getText() );
}
newTransaction.commit();
newSession.close();
```

文本串“from Message as m order by m.text asc”是一个使用Hibernate自己的查询语言（HQL）表示的查询。当调用find()方法时，这个查询在内部被转化成下面的SQL语句：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
order by m.MESSAGE_TEXT asc
```

下面的消息会被打印出来：

```
1 message(s) found:
Hello World
```

如果以前你从没有用过像Hibernate这样的工具，你可能期望在代码或元数据里看到SQL语句。但它们并不存在。所有的SQL都是在运行时（对所有可重用的SQL语句实际是在起动时）生成的。

为了允许这个魔法发生，Hibernate需要更多关于Message类如何被持续化的信息。这些信息通常在XML映射文件里提供。这个映射文件定义了Message类的属性如何映射到MESSAGES表的字段和其它一些信息。让我们看一下清单2.2中的映射文件：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class
        name="hello.Message"
        table="MESSAGES">
        <id
            name="id"
            column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property
            name="text"
            column="MESSAGE_TEXT"/>
        <many-to-one
            name="nextMessage"
            cascade="all"
            column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

（清单2.2）

这个映射文件告诉Hibernate：Message类被持续化到MESSAGES表里，标识符属性映射到名为MESSAGE_ID的字段，文本属性映射到名为MESSAGE_TEXT的字段，名为nextMessage的属性是一个多对一的关联，它映射到名为NEXT_MESSAGE_ID的字段（现在不用担心其它细节）。

像你看到的一样，XML文件并不难以理解。你可以很容易地进行手工编写和维护。第3章，我们将讨论一种根据嵌入在源代码中的注释生成XML文件的方式。无论你选择了哪种方法，Hibernate都有足够的信息完全地生成插入、更新、删除和选取Message实例所需的所有的SQL语句。你不再需要手工编写这些SQL语句。

注意 许多Java开发者都在抱怨伴随着J2EE开发的元数据的“地狱”。许多人建议我们远离XML元数据，返回到普通的Java代码。虽然对有些问题我们赞同这个建议，但ORM代表了一种确实需要基于文本的元数据的情形。Hibernate有一些明显的缺省值这减少了你的键入，并且它还有一个成熟的文档类型定义，可以用于在编辑器中进行自动地生成或验证。你甚至可以使用各种工具自动地生成元数据。

现在，让我们改变一下我们的第一个message，并且在其上创建一个新的关联message，参见清单2.3。

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
// 1 是第一个消息生成的ID
Message message =
        (Message) session.load( Message.class, new Long(1) );
message.setText("Greetings Earthling");
Message nextMessage = new Message("Take me to your leader (please)");
message.setNextMessage( nextMessage );
tx.commit();
session.close();
```

（清单2.3）

这段代码在同一个事务里调用了三条SQL语句：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES
set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

注意Hibernate如何检测到第一个消息的text和nextMessage属性的修改并且自动更新了数据库。我们已经利用了Hibernate称为自动脏检查的特征：当我们在一个事务里修改了对象的状态时，这个特征节省了我们明确地要求Hibernate更新数据库的努力。相似地，当在第一个消息对象上引用一个新消息对象时，你可以看到新消息对象被持续化了。这个特征被称为级联保存：只要对一个已持续化的实例是可达的，就能够节省我们明确地调用save方法对新对象进行持续化的努力。也要注意到SQL语句的顺序不同于我们设置属性值的顺序。Hibernate使用了一个复杂的算法来确定一个有效的顺序以避免违反数据库的外键约束，但对用户来说，这仍然是充分可预知的。这个特征被称为事务写置后。

如果我们再次运行“Hello World”，它会打印出：

```
2 message(s) found:
Greetings Earthling
Take me to your leader (please)
```

这是目前为止我们完成的“Hello World”应用。现在我们终于有了一些代码经历，我们将后退一步介绍Hibernate主要的API预览。

2．2 理解Hibernate的体系结构

为了在你的应用中使用Hibernate，编程接口是你首先需要学习的东西。API设计的一个主要目标是保持软件组件间的接口尽可能小。然而，实际上ORM的API不会特别少。不过也不用担心，你不必马上理解Hibernate的所有接口。图2.1介绍了在业务层和持续层中最重要的一些Hibernate接口的角色。我们将业务层显示在持续层之上，因为在传统的分层应用中业务层担当持续层的一个客户。注意一些简单的应用可能不会明确区分业务层与持续层；这也没有问题——它只是将图形进行了简化。

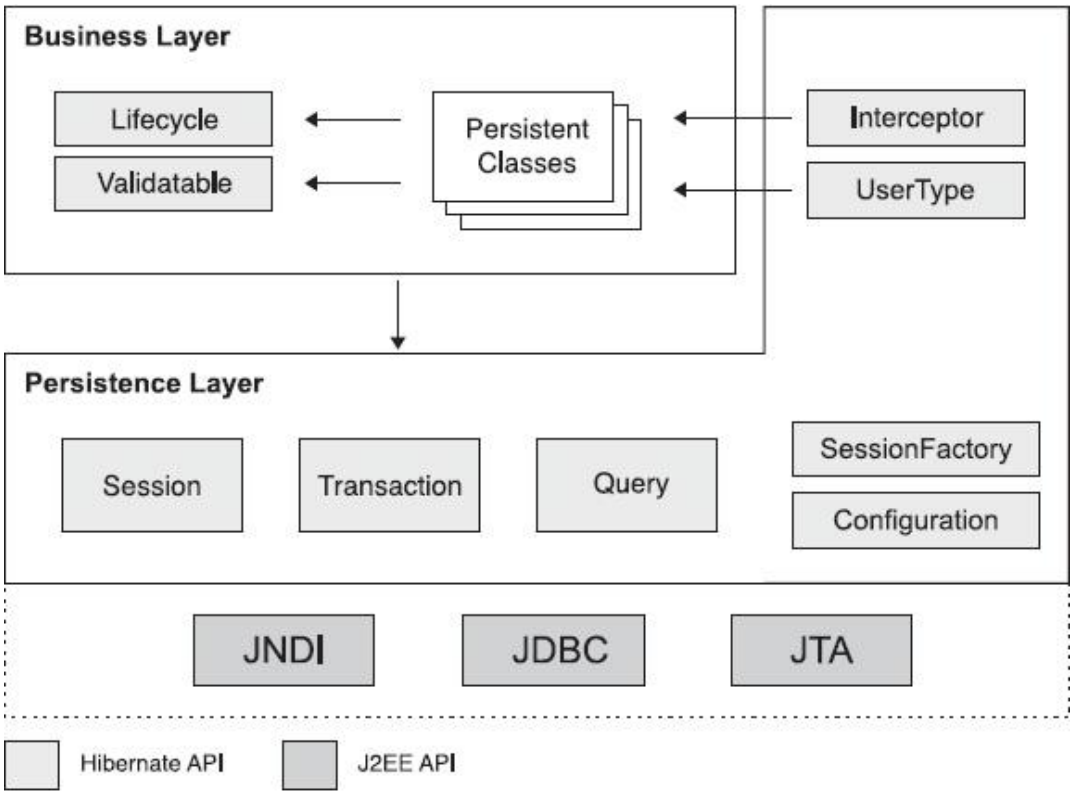

Figure 2.1 High-level overview of the Hibernate API in a layered architecture

（图2.1）

图2.1中显示的接口可以近似地分为如下几类：

■ 由应用调用以完成基本的CRUD和查询操作的接口。这些接口是应用的业务/控制逻辑对Hibernate的主要的依赖点。它们包括Session，Transaction和Query。

■ 由应用的底层代码调用以配置Hibernate的接口，最重要的是Configuration类。

■ 允许应用对Hibernate内部出现的事件进行处理的回调接口，例如Interceptor，Lifecycle和Validatable接口。

■ 允许对Hibernate强大的映射功能进行扩展的接口，例如UserType，CompositeUserType和IdentifierGenerator。这些接口由应用的底层代码实现（如果需要的话）。

Hibernate使用了许多现有的Java API，包括JDBC，Java事务API（JTA）和Java命名和目录接口（JNDI）。JDBC为关系数据库的共通功能提供了一个基本级别的抽象，允许几乎所有具有JDBC驱动的数据库被Hibernate支持。JNDI和JTA允许Hibernate与J2EE应用服务器进行集成。

在本节中，我们并不包含Hibernate API方法的详细语义，只是介绍每一个主要接口的角色。你可以在net.sf.hibernate包中找到这些接口中的大部分。让我们依次简单地看一下每一个接口。

2．2．1 核心接口

有五个核心接口几乎在每个Hibernate应用中都会用到。使用这些接口，你可以存储与取出持续对象或者对事务进行控制。

Session接口

Session（会话）接口是Hibernate应用使用的主要接口。会话接口的实例是轻量级的并且创建与销毁的代价也不昂贵。这很重要因为你的应用可能始终在创建与销毁会话，可能每

一次请求都会如此。Hibernate会话并不是线程安全的因此应该被设计为每次只能在一个线程中使用。

Hibernate会话是一个介于连接和事务之间的概念。你可以简单地认为会话是对于一个单独的工作单元已装载对象的缓存或集合。Hibernate可以检测到这个工作单元中对象的改变。我们有时也将会话称为持续性管理器，因为它也是与持续性有关的操作例如存储和取出对象的接口。注意，Hibernate会话与Web层的HttpSession没有任何关系。当我们在本书中使用会话时，我们指的是Hibernate会话。为了区别，有时我们将HttpSession对象称为用户会话。

我们将在第4章第4.2节“持续性管理器”中详细地描述会话接口。

SessionFactory接口

应用从SessionFactory（会话工厂）里获得会话实例。与会话接口相比，这个对象不够令人兴奋。

会话工厂当然不是轻量级的！它打算在多个应用线程间进行共享。典型地，整个应用只有唯一的一个会话工厂——例如在应用初始化时被创建。然而，如果你的应用使用Hibernate访问多个数据库，你需要对每一个数据库使用一个会话工厂。

会话工厂缓存了生成的SQL语句和Hibernate在运行时使用的映射元数据。它也保存了在一个工作单元中读入的数据并且可能在以后的工作单元中被重用（只有类和集合映射指定了这种二级缓存是想要的时才会如此）。

Configuration接口

Configuration（配置）对象用来配置和引导Hibernate。应用使用一个配置实例来指定映射文件的位置和Hibernate的特定属性，然后创建会话工厂。

即使配置接口只担当了整个Hibernate应用范围内一个相对较小的部分，但它却是在你开始使用Hibernate时遇到的第一个对象。第2.3节比较详细地介绍了一些配置Hibernate的问题。

Transaction接口

Transaction（事务）接口是一个可选的API。Hibernate应用可以选择不使用这个接口，而是在它们自己的底层代码中管理事务。事务将应用代码从下层的事务实现中抽象出来——这可能是一个JDBC事务，一个JTA用户事务或者甚至是一个公共对象请求代理结构（CORBA）——允许应用通过一组一致的API控制事务边界。这有助于保持Hibernate应用在不同类型的执行环境或容器中的可移植性。

我们自始至终在本书中使用Hibernate事务API。事务与事务接口在第5章进行说明。

Query与Criteria接口

Query（查询）接口允许你在数据库上执行查询并控制查询如何执行。查询使用HQL或者本地数据库的SQL方言编写。查询实例用来绑定查询参数，限定查询返回的结果数，并且最终执行查询。

Criteria（标准）接口非常小，它允许你创建和执行面向对象的标准查询。

为了帮助应用代码减少冗余，Hibernate在会话接口上提供了一些快捷方法，允许你可以在一行代码内调用一个查询。在本书中我们不使用这些快捷方法，相反，我们会一直使用查询接口。

查询实例是轻量级的并且不能在创建它的会话外使用。我们将在第7章描述查询接口的这个特征。

2．2．2 回调接口

当一个对象发生了应用感兴趣的事情——例如，当一个对象被装载、保存或删除时，回调接口允许应用可以接收到通知。Hibernate应用并不必需实现这些回调，但是在实现特定类型的功能（例如创建审计记录）时，它们非常有用。

接口Lifecycle和Validatable允许持续对象对与其有关的生命周期事件做出反应。持续性生命周期由对象的CRUD操作构成。Hibernate开发组受到了其它具有相似回调接口的ORM解决方案的很深的影响。后来，他们认识到让持续类实现Hibernate的特定接口可能不是一个好主意，因为这样做污染了我们的持续类使其成为了不可移植的代码。因此这些方法不再赞成使用，我们也不在本书中讨论它们。

引入接口Interceptor是为了允许应用处理回调而又不用强制持续类实现Hibernate特定的API。接口Interceptor的实现被作为参数传递给持续类的实例。我们将在第8章讨论一个这样的例子。

2．2．3 类型

一个基础的并且非常强大的体系结构元素是Hibernate的类型的概念。Hibernate的类型对象将一个Java类型映射到数据库字段的类型（实际上，类型可能跨越多个字段）。持续类所有的持续属性，包括关联，都有一个对应的Hibernate类型。这种设计使Hibernate变得极端灵活并易于扩展。

内建类型的范围非常广泛，覆盖了所有的Java基础类型和许多JDK类，包括java.util.Currency，java.util.Calendar，byte[]和java.io.Serializable。

甚至更好一些，Hibernate支持用户自定义类型。它提供了UserType和CompositeUserType

接口允许你增加自己的类型。使用这个特征，应用使用的共通类例如Address，Name或MonetaryAmount就可以方便优雅地进行处理了。自定义类型被认为是Hibernate的重要特征，并鼓励你对它们进行新的或创造性的使用。

我们将在第6章第6.1节“理解Hibernate的类型系统”中介绍Hibernate类型和用户自定义类型。

2．2．4 扩展接口

Hibernate提供的大多数功能都是可配置的，允许你在一些内置的策略中进行选择。当内置策略不能满足需要时，Hibernate通常会允许你通过实现一个接口插入你自己的定制实现。扩展点包括：

- 主键生成（IdentifierGenerator接口）
- SQL方言支持（Dialect抽象类）
- 缓存策略（Cache和CacheProvider接口）
- JDBC连接管理（ConnectionProvider接口）
- 事务管理（TransactionFactory，Transaction和TransactionManagerLookup接口）
- ORM策略（ClassPersister接口层次）
- 属性访问策略（PropertyAccessor接口）
- 代理创建（ProxyFactory接口）

对上面列出的每一个接口Hibernate至少都自带了一种实现，因此如果你想对内置功能进行扩展的话通常不需要你从头开始。对于自己的实现，你可以将源代码作为例子来使用。

现在你可以看到在我们开始使用Hibernate编写任何代码之前我们必须回答这个问题：我们如何取得一个需要的会话呢？

2．3 基本配置

我们已经看了一个应用的例子并且分析了Hibernate的核心接口。为了在应用中使用Hibernate，你必须知道如何配置它。Hibernate可以配置在几乎所有的Java应用和开发环境中运行。通常，Hibernate在两层或三层的客户/服务器应用中使用，而且只配置在服务器端。客户端应用通常是一个Web浏览器，Swing和SWT的客户端应用并不常见。虽然在本书中我们集中在多层的Web应用上，我们的说明也可以等同地应用到其它体系结构上，例如命令行应用。理解在管理与非管理环境中配置Hibernate的区别是非常重要的：

■ 管理环境——将资源例如数据库连接进行池化并且允许事务边界和安全通过声明进行指定（即使用元数据）。J2EE应用服务器例如JBoss，BEA WebLogic或IBM Websphere都实现了标准的管理环境（J2EE特定的）。

■ 非管理环境——通过线程池提供了基本的并发管理。像Jetty或Tomcat这样的Servlet容器为Java Web应用提供了非管理的服务器环境。独立的桌面或命令行应用也被认为是非管理的。非管理环境并不提供自动的事务、资源管理和底层的安全结构。应用自己管理数据库连接并划分事务界限。

Hibernate试图抽象它的配置环境。在非管理环境的场合，Hibernate自己处理事务和JDBC连接（或者委托应用代码处理这些问题）。在管理环境的场合，Hibernate与容器管理的事务和数据源相结合。在两种环境中都可以根据配置来设定Hibernate。

在管理与非管理两种环境中，你必须作的第一件事都是起动Hibernate。实际上，这非常简单。你只要根据配置创建一个会话工厂即可。

2．3．1 创建会话工厂

为了创建一个会话工厂，在应用初始化时你首先要创建一个单独的配置实例并使用它设置映射文件的位置。一旦配置完成，你就可以使用配置实例创建会话工厂了。在会话工厂创建以后，你可以丢弃配置类。

下面的代码起动了Hibernate：

```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties( System.getProperties() );
SessionFactory sessions = cfg.buildSessionFactory();
```

映射文件（Message.hbm.xml）的位置是相对于应用的classpath的。例如，如果classpath是当前目录，则Message.hbm.xml文件必须位于hello目录下。XML映射文件必须位于classpath里。在这个例子里，我们也使用了虚拟机的系统属性来设置其它的配置选项（这些选项可能早已被应用代码或者作为起动选项设置过了）。

方法链 方法链是许多Hibernate接口支持的编程风格。这种风格在Smalltalk中比在Java中更流行，但是许多人认为它与普通的Java风格相比更不易阅读并且更难于调试。然而，在大多数情况下，它都非常方便。

大多数Java开发者都将设置或计算方法的类型设置为void，这意味着它们没有返回值。在Smalltalk里没有void类型，设置或计算方法通常将接收的对象返回。这就允许我们如下重写前面例子的代码：

```
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

注意我们不再需要为配置类声明局部变量了。我们在一些代码例子中使用了这种风格；

但如果你不喜欢，你自己并不需要使用。如果你确实使用了这种编码风格，最好将每个方法调用写在不同的行上。否则，在调试器里单步调试代码可能会比较困难。

依照惯例，Hibernate的XML映射文件以.hbm.xml扩展名命名。另一个惯例是每个类一个映射文件，而不是将所有的映射列在一个文件中（这是可能的但被认为是很差的风格）。我们的例子“Hello World”只有一个持续类，但假定我们有多个持续类，并且每个类一个XML映射文件，我们应该将这些映射文件放在哪里呢？

Hibernate文档推荐每个持续类的映射文件放在与映射类相同的目录中。例如，Message类的映射文件应该放在hello目录中，并且命名为Message.hbm.xml。如果我们有另一个持续类，它应该被定义在自己的映射文件中。我们建议你遵循这项实践。许多框架鼓励单一的元数据文件，例如在struts中建立的struts-config.xml，就是“元数据地狱”的一个主要贡献者。每当需要时你可以调用addResource()装载多个映射文件，或者，如果你遵循刚刚描述过的惯例，你可以使用addClass()方法，传递一个持续类作为参数：

```
SessionFactory sessions = new Configuration()
    .addClass(org.hibernate.auction.model.Item.class)
    .addClass(org.hibernate.auction.model.Category.class)
    .addClass(org.hibernate.auction.model.Bid.class)
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

AddClass()方法假定映射文件的文件名以.hbm.xml扩展名结尾并且与被映射的类文件配置在一起。

我们已经展示了单独的会话工厂的创建，它是大多数应用需要的全部内容。如果你需要另一个会话工厂——例如，有多个数据库的情形——你只需要重复这个过程即可。每个会话工厂仅用于一个数据库，并且准备生成会话，用来与特定的数据库和一组类映射一起工作。

当然，还有更多的内容需要配置而不仅仅是指明映射文件。你也需要指定如何获得数据库连接，以及其它各种在运行时影响Hibernate行为的设置。多数配置属性可能会压倒性的出现（在Hibernate文档中包含一个完全的列表），但是不用担心，大多数属性都定义了合理的缺省值，并且只有少数是普遍需要的。

为了指定配置选项，你可以使用下列任何技术：

- 传递一个java.util.Properties实例到Configuration.setProperties()。
- 使用java -Dproperty=value设置系统属性。
- 将一个名为hibernate.properties的文件放置在classpath中。
- 在classpath中的hibernate.cfg.xml文件里包含<property>元素。

第一、二个选项除了快速测试和原型外很少使用，大多数应用需要一个固定的配置文件。文件hibernate.properties和hibernate.cfg.xml提供了配置Hibernate的相同的功能。选择使用哪个文件依赖于你的语法爱好。像你将要在本章中看到的一样，也可能混合使用这两种选择并且对开发和配置使用不同的设置。

一种罕见的使用选择是允许应用从会话工厂中打开一个Hibernate会话时提供JDBC连接（例如通过调用sessions.openSession(myConnection)）。使用这种选择意味着你不必指定任何数据库连接属性。对新应用我们不推荐这种方法，因为它们可以使用环境中的数据库连接的底层构造（例如，JDBC连接池或者应用服务器数据源）进行配置。

在所有的配置选项中，数据库连接的设置是最重要的。它们在管理与非管理环境中的设置是不同的，因此我们独立地处理这两种情况。让我们首先从非管理环境开始。

2．3．2 在非管理环境中配置

在非管理环境例如Servlet容器中，应用负责取得JDBC连接。Hibernate是应用的一部分，因此，它会负责取得这些连接。但是需要你告诉Hibernate如何取得（或创建）JDBC连接。通常，每次与数据库交互都创建一个连接的做法是不可取的。相反，Java应用应该使用JDBC连接池。这有三方面的原因：

- 获得新连接的代价是很昂贵的。
- 维护许多无用的连接也是很浪费的。
- 对许多驱动程序来说创建预编译语句同样是很昂贵的。

图2.2显示了JDBC连接池在Web应用运行环境中的角色。因为非管理环境没有实现连接池，所以应用必须实现自己的池化算法或者依赖于第三方类库例如开源的C3P0连接池。不使用Hibernate时，应用代码通常直接调用连接池来获得JDBC连接并执行SQL语句。

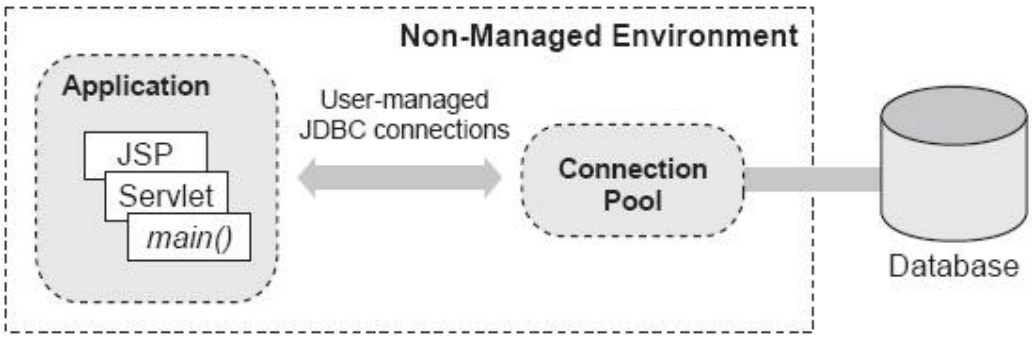

Figure 2.2 JDBC connection pooling in a non-managed environment

（图2.2）

使用Hibernate时，图像发生了改变：它被作为JDBC连接池的一个客户，如图2.3所示。应用代码使用Hibernate会话和查询API进行持续性操作并且理论上只需要使用Hibernate事务API管理数据库事务。

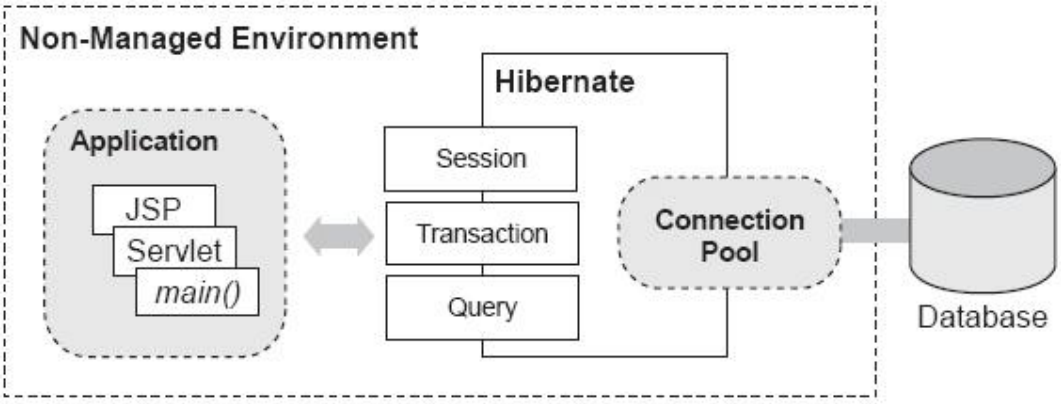

Figure 2.3 Hibernate with a connection pool in a non-managed environment

（图2.3）

使用连接池

Hibernate定义了一个插件体系结构，允许你与任何连接池集成。然而，Hibernate内置了对C3P0的支持，因此我们将使用它。Hibernate使用特定的属性为你建立了连接池。一个使用了C3P0的hibernate.properties文件的例子参见清单2.4。

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size = 5
hibernate.c3p0.max_size = 20
hibernate.c3p0.timeout = 300
hibernate.c3p0.max_statements = 50
hibernate.c3p0.idle_test_period = 3000
```

（清单2.4）

从第一行开始，以上代码分别指定了下面这些信息：

■ 实现JDBC驱动程序的Java类的名称（驱动程序的jar文件必须位于应用的classpath中）。

■ 指定JDBC连接的主机名和数据库名的JDBC URL。

■ 数据库用户名。

■ 指定用户的数据库密码。

■ 一种数据库方言。尽管有ANSI标准化的努力，不同的数据库厂商还是有不同的SQL实现。因此，你必须指定一种方言。Hibernate对所有流行的SQL数据库都包含内建的支持，并且也很容易定义新的方言。

■ C3P0保持的JDBC连接的最小值。

■ 池中连接的最大值。如果在运行时这个值被耗尽则会抛出一个异常。

■ 一个超时时间（在本例中，是5分钟或300秒），在这之后空闲的连接会被从池中删除。

■ 可以被缓存的预编译语句的最大数量。对预编译语句进行缓存是高性能使用Hibernate必需的。

■ 连接自动生效前的空闲时间（以秒为单位）。

以hibernate.c3p0.*格式指定的属性选择了C3P0作为Hibernate的连接池（为了能支持C3P0，你不需要再设置任何其它选项）。C3P0有比我们在前面的例子中展示的更多的特征，因此我们建议你查询Hibernate API文档。类net.sf.hibernate.cfg.Environment的Javadoc文档化了Hibernate的每一个配置属性，包括所有C3P0相关的设置和其它Hibernate直接支持的第三方连接池的设置。

其它支持的连接池是Apache DBCP和Proxool。在你决定之前，你应该首先在你的环境里试验一下每一个连接池。不过，Hibernate社团更倾向于C3P0和Proxool。

Hibernate也包含了一个缺省的连接池机制。这个连接池只适用于测试或试验Hibernate。在产品系统中，你不应该使用这个内建的池。在并发请求较多的环境中它的设计是不可扩展的，并且它缺乏一些专业的连接池才具有的容错特征。

起动Hibernate

如何使用这些属性起动**Hibernate**？你在一个名为**hibernate.properties**的文件中声明这些属性，因此你只需要将这个文件放在应用的**classpath**中即可。当你创建了一个配置对象，**Hibernate**首次进行初始化时，它会被自动地检测和读取。

让我们总结一下目前为止你已经学到的配置步骤（如果你想在非配置环境下继续的话，这是下载和安装**Hibernate**的一个很好的时机）。

1．下载并解包你的数据库的**JDBC**驱动程序，这通常可以从数据库厂商的**Web**站点得到。将**JAR**文件放到应用的**classpath**中；同时也将**hibernate2.jar**放在相同的位置。

2．将**Hibernate**的依赖包放到**classpath**中；它们在**lib/**目录中与**Hibernate**一起发布。同时也要看一下**lib/**目录下的文本文件**README.txt**，它包含一个必需与可选库的列表。

3．选择一个**Hibernate**支持的**JDBC**连接池并使用属性文件进行配置。不要忘记指定**SQL**方言。

4．通过将它们放在一个位于**classpath**中的**hibernate.properties**文件中，让配置对象知道这些属性。

5．在你的应用中创建一个配置对象的实例并使用**addResource()**或**addClass()**方法装载**XML**映射文件。通过调用配置对象的**buildSessionFactory()**方法创建一个会话工厂。

很不幸，你还没有任何映射文件。如果你喜欢，你可以运行“**Hello World**”的例子或者跳过本章剩余的部分并开始学习第**3**章持续类与映射。或者，如果你想知道更多关于在管理环境中使用**Hibernate**的知识，请继续往下读。

2．3．3 在管理环境中配置

管理环境处理特定的“**cross-cutting**”关系，例如应用安全（授权与验证），连接池和

事务管理。J2EE应用服务器是典型的管理环境。虽然应用服务器通常是为了支持EJB而设计的，但即使你不使用EJB实体Bean，你仍然可以利用它提供的其它服务。

Hibernate经常与会话或消息驱动EJB一起使用，如图2.4所示。像servlet、JSP和独立的应用一样，EJB调用相同的Hibernate API：Session（会话）、Transaction（事务）和Query（查询）。与Hibernate相关的代码在非管理与管理环境之间是完全可移植的。Hibernate透明地处理了不同的连接与事务策略。

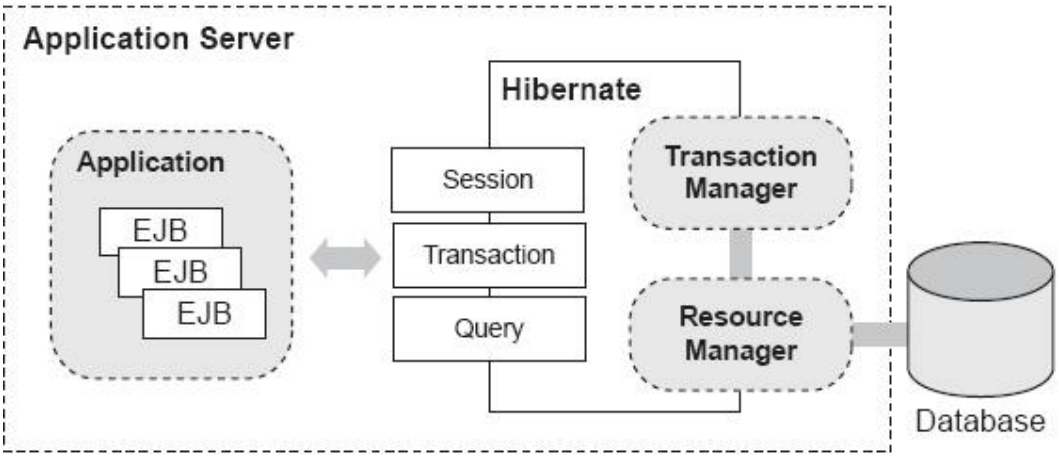

Figure 2.4 Hibernate in a managed environment with an application server

（图2.4）

应用服务器将连接池对外显示为JNDI绑定数据源，它是javax.jdbc.Datasource类的一个实例。你需要提供一个JNDI全限定名来告诉Hibernate，到哪里去查找JNDI数据源。这种情况下的一个Hibernate配置文件的例子参见清单2.5。

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

（清单2.5）

这个文件首先给出了数据源的JNDI名。数据源必须在J2EE企业应用的配置描述符中进行配置；这是一个厂商特定的设置。接着，你允许Hibernate与JTA集成。为了与容器事务完全集成， Hibernate需要定位应用服务器的事务管理器。J2EE规范没有定义标准的方法，但是Hibernate包含了对所有流行的应用服务器的支持。当然，最后还需要指定SQL方言。

现在你已经正确地配置了每件事情，在管理环境中使用Hibernate与在非管理环境中没有太多的区别：都只是使用映射创建一个配置对象并构造一个会话工厂。然而，许多与事物环境有关的设置值得进行额外的考虑。

Java早就有了一个标准的事务API：JTA，它用来在J2EE管理环境中控制事务。这被称作容器管理的事务（CMT）。如果存在JTA事务管理器，JDBC连接将被它支持并完全位于它的控制之下。这与非管理环境不同，在非管理环境中应用（或连接池）直接管理JDBC连接和JDBC事务。

因此，管理环境与非管理环境可以使用不同的事务方法。因为Hibernate需要在这两种环境之间保证可移植性，因此它定义了一组控制事务的API。Hibernate的事务接口抽象了下层的JTA或JDBC事务（或者，甚至潜在的CORBA事务）。这个下层的事务策略可以使用属性hibernate. connection.factory_class来设置，它可以取下列两值之一：

■ net.sf.hibernate.transaction.JDBCTransactionFactory代表了直接的JDBC事务。这种策略应该与非管理环境中的连接池一起使用，并且如果没有指定任何策略时它就是缺省值。

■ net.sf.hibernate.transaction.JTATransactionFactory代表了JTA。对于CMT这是正确的策略，此时连接由JTA支持。注意如果调用beginTransaction()时已经有一个JTA事务在

运行，随后的工作将发生在那个事务的上下文中（否则，将会开始一个新的JTA事务）。

对Hibernate事务API更详细的介绍与它对特定应用场景的影响，请参考第5章第5.1节“事务”。你只要记住使用J2EE服务器工作时必需的两个步骤：像前面描述的那样为Hibernate事务API设置工厂类以支持JTA，并且声明你的应用服务器特定的事务管理器的查找策略。只有你使用了Hibernate的二级缓存系统时查找策略才是必需的，但即使你没有使用缓存设上它也没有坏处。

Hibernate与Tomcat Tomcat并不是一个完整的应用服务器；它只是一个Servlet容器，尽管它包含一些通常只有在应用服务器中才能找到的特征。其中的一个特征可能被Hibernate使用：Tomcat的连接池。Tomcat内部使用的是DBCP连接池，但也像真正的应用服务器一样将它作为JNDI数据源对外显示。为了配置Tomcat数据源，你需要根据Tomcat JNDI/JDBC文档的指导编辑server.xml文件。你可以设置hibernate.connection.datasource来配置Hibernate使用这个数据源。注意Tomcat并不包含一个事务管理器，因此这种情形更像以前描述的非管理环境。

无论使用的是一个简单的Servlet容器还是一个应用服务器，现在你应该有了一个正在运行的Hibernate系统。创建并编译一个持续类（例如，最初的Message类），将Hibernate和它需要的类库以及文件hibernate.properties拷贝到classpath中，构造一个会话工厂。

下一节包含高级的Hibernate配置选项。其中许多是推荐使用的，例如将可执行的SQL语句写入日志以利于调试；或者使用方便的XML配置文件而不是无格式的属性。然而，你可以安全的跳过这一节而直接去读第三章，在你学到更多关于持续类的知识后再回来。

2．4 高级配置设置

当你最终有了一个可以运行的Hibernate应用时，对Hibernate所有的配置参数有一个全面的了解是非常值得的。这些参数允许你优化Hibernate运行时的行为，特别是调整与JDBC的交互（例如，使用JDBC进行批处理更新时）。

现在我们不会让你对这些细节感到厌烦；关于配置选项最好的信息来源当然是Hibernate的参考文档。在上一节里，我们已经向你说明了一些开始时需要知道的选项。

然而，有一个参数我们必须在这里强调一下。无论何时你使用Hibernate开发软件时都会频繁地用到它。将属性hibernate.show_sql设置为true就可以将所有生成的SQL记录到控制台上。你可以使用它来诊断故障，调整性能或者只是看看生成了什么。它有利于你知道ORM层正在做什么——这就是为什么ORM不对开发者隐藏SQL的原因。

到目前为止，我们一直假定你在使用hibernate.properties文件或者在程序里使用一个java.util.Properties的实例来指定配置参数。除此之外，还有你可能会喜欢的第三种选择：使用XML配置文件。

2．4．1 使用基于XML的配置

你可以使用一个XML配置文件（如清单2.6所示）来完全地配置一个会话工厂。不像只包含配置参数的文件hibernate.properties，文件hibernate.cfg.xml也可以指定映射文件的位置。许多用户更喜欢以这种方式集中Hibernate的配置，而不是在应用代码中往配置对象里增加参数。

```
<?xml version='1.0'encoding='utf-8'?>
<!DOCTYPE hibernate-configuration                                    ①
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
```

```
    <session-factory name="java:/hibernate/HibernateFactory">                    ②
        <property name="show_sql">true</property>
        <property name="connection.datasource">                                  ③
            java:/comp/env/jdbc/AuctionDB
        </property>
        <property name="dialect">
            net.sf.hibernate.dialect.PostgreSQLDialect
        </property>
        <property name="transaction.manager_lookup_class">
            net.sf.hibernate.transaction.JBossTransactionManagerLookup
        </property>
        <mapping resource="auction/Item.hbm.xml"/>                               ④
        <mapping resource="auction/Category.hbm.xml"/>
        <mapping resource="auction/Bid.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

（清单2.6）

① 文档类型声明，XML解析器根据它声明的Hibernate配置DTD来验证文档。

② 可选的name属性等价于属性hibernate.session_factory_name，它用于会话工厂的JNDI绑定，将在下一节进行讨论。

③ 可以不使用“hibernate”作为前缀指定的Hibernate属性。属性名与值在其它方面与程序化的配置属性是相同的。

④ 映射文件可以作为应用资源甚至是硬编码的文件名来指定。这里使用的文件来自于我们的在线拍卖应用，我们将在第3章进行介绍。

现在你可以使用下面的语句初始化Hibernate了：

```
SessionFactory sessions = new Configuration()
    .configure().buildSessionFactory();
```

稍等——Hibernate如何知道配置文件位于哪里？

当调用configure()方法时，Hibernate在classpath中查找名为hibernate.cfg.xml的文件。如果你想使用一个不同的文件名或者想让Hibernate在一个子目录中进行查找，你必须将路径传递给configure()方法。

```
SessionFactory sessions = new Configuration()
    .configure("/hibernate-config/auction.cfg.xml")
    .buildSessionFactory();
```

使用XML配置文件的确比属性文件甚至是程序化的属性配置让人感觉更舒适。这种方法主要的好处是可以让类映射文件从应用的源代码中（即使它仅出现在起动时的帮助类中）具体地表示出来。举例来说，你可以对不同的数据库和环境（开发或产品）使用不同的映射文件（和不同的配置选项），并通过程序切换它们。

如果在classpath中同时存在hibernate.properties和hibernate.cfg.xml两个文件，则XML配置文件的设置会重载属性文件。如果你想在属性文件中保持一些基本设置并且在每个部署中使用XML配置文件重载它们时这非常有用。

你可能已经注意到了在XML配置文件中同时给会话工厂指定了一个名字。在会话工厂创建之后，Hibernate自动地使用这个名字将它绑定到JNDI。

2．4．2 绑定到JNDI的会话工厂

在大多数Hibernate应用中，在应用初始化期间会话工厂都会被实例化一次。然后这个唯一的实例会被一个特定过程的所有代码使用，并且所有的会话都由这个唯一的会话工厂创建。一个经常询问的问题是这个工厂应该放到什么地方并且如何轻而易举地访问它。

在J2EE环境中，绑定到JNDI的会话工厂可以很容易地在不同的线程和各种支持Hibernate的组件之间进行共享。当然，JNDI并不是应用组件能够获得会话工厂的唯一方式。这种注册

模式有多种可能的实现，包括使用ServletContext或者单体中的static final变量。一种特别优雅的方法是使用一个应用范围的IoC（控制反转）框架组件。然而，JNDI是一种比较流行的方法（稍后你将看到，它作为JMX服务对外显示）。我们将在第8章第8.1节“设计分层的应用”中讨论一些其它的可选方案。

注意 Java命名与目录接口（JNDI）API允许从一个层次结构（目录树）中存储与取出对象。JNDI实现了注册模式。底层对象（事务上下文，数据源），配置设置（环境设置，用户注册）甚至是应用对象（EJB引用，对象工厂）都可以绑定到JNDI。

如果为属性hibernate.session_factory_name设置了目录节点的名称，会话工厂会自动地将它自己绑定到JNDI。如果你的运行环境没有提供一个缺省的JNDI上下文（或者缺省的JNDI实现不支持Referenceable的实例），你需要使用属性hibernate.jndi.url和hibernate.jndi.class指定JNDI的初始上下文。

这儿有一个Hibernate配置的例子，它使用Sun的（免费的）基于文件系统的JNDI实现（fscontext.jar），将会话工厂绑定到了名称hibernate/HibernateFactory上：

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = hibernate/HibernateFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

当然，对这个任务你也可以使用基于XML的配置。这个例子也不太现实，因为大多数通过JNDI提供连接池的应用服务器都会有一个JNDI的可写的缺省上下文的实现。JBoss当然也有，

因此你可以跳过最后两个属性仅指定一个会话工厂名即可。现在你需要做的所有的事情是调用一次Configuration.configure().buildSessionFactory()来对绑定进行初始化。

注意 Tomcat带有一个只读的JNDI上下文，在Servlet容器起动之后对应用级别的代码来说它是不可写的。Hibernate不能绑定到这种类型的上下文上；你必须或者使用一个完全的上下文的实现（例如Sun的文件系统上下文）或者忽略配置属性session_factory_name来禁用会话工厂的JNDI绑定。

让我们看一些非常重要的用于记录Hibernate操作日志的其它的配置设定。

2．4．3 日志

Hibernate（与许多其它的ORM实现）异步地执行SQL语句。通常当应用调用Session.save()时并不会执行INSERT语句，当应用调用Item.addBid()时也不会立即执行UPDATE语句。相反，SQL语句通常都在事务的终点执行。这种行为被称作写置后，我们以前曾经提到过。

有时当跟踪与调试ORM代码很重要时，这个事实也会变得很明显。理论上，应用将Hibernate作为黑盒看待而忽略它的行为是可能的。当然Hibernate应用不会觉察到这种异步（至少不对直接的JDBC调用进行重新排序时不会）。然而，当你遇到一个难题时，你需要能够精确地看到Hibernate内部发生了什么。因为Hibernate是开源的，你可以很容易地单步调试Hibernate的代码。有时这样做很有帮助！但是，特别是面对异步行为时，调试Hibernate会很快地让你迷失。此时，你可以使用日志来得到Hibernate的内部视图。

我们早已提到过hibernate.show_sql这个配置参数，当遇到麻烦时它通常是你能想到的第一个入口。有时只有SQL是不够的，那时，你必须钻得更深一些。

Hibernate使用Apache的commons-logging将所有感兴趣的事件记入了日志。

commons-logging是很薄的一个抽象层，它直接输出到Apache的log4j里（如果你在classpath里放入了log4j.jar）或者是JDK1.4的logging里（如果你运行在JDK1.4以上并且log4j不存在时）。我们推荐log4j，因为它更成熟，更流行，并且开发也更活跃。

为了看到log4j的输出，你需要将一个名为log4j.properties的文件放到classpath里（紧挨着hibernate.properties或者hibernate.cfg.xml）。下面的例子指示将所有日志信息输出到控制台上：

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
⇛ %5p %c{1}:%L - %m%n
### root logger option ###
log4j.rootLogger=warn, stdout
### Hibernate logging options ###
log4j.logger.net.sf.hibernate=info
### log JDBC bind parameters ###
log4j.logger.net.sf.hibernate.type=info
### log PreparedStatement cache activity ###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

使用这个配置，你不会在运行时看到太多的日志信息。将log4j.logger.net.sf.hibernate的类型由info替换为debug将会显示出Hibernate的内部工作状况。确保你在产品环境中不会这么做——写日志比实际的数据库访问要慢很多。

最终，你有了hibernate.properties，hibernate.cfg.xml和log4j.properties三个配置文件。

如果你的应用服务器支持Java管理扩展的话，还有另外一种配置Hibernate的方式。

2．4．4 Java管理扩展（JMX）

在Java世界里充满了规范、标准当然也包括它们的实现。一个相对较新但又非常重要的标准有了它的初版：Java管理扩展（JMX）。JMX是关于系统组件或者更好一点是关于系统服务的管理。

Hibernate怎样适应这种新的情形？当配置到应用服务器中时，Hibernate使用了像受管理的事务与池化的数据库事务等一些其它的服务。但是为什么不让Hibernate本身成为一个受管理的服务，让其它服务可以依赖或使用它？使用Hibernate的JMX集成，使Hibernate成为一个受管理的JMX组件是可以做到这一点的。

JMX规范定义了下面的组件：

- JMX MBean——揭示了管理接口的可重用组件（通常是底层结构）
- JMX容器——协调对MBean的类属的访问（本地或远程的）
- JMX客户（通常是类属的）——可用于通过容器管理MBean

一个支持JMX的应用服务器（例如JBoss）担当JMX容器并允许MBean作为应用服务器起动过程的一部分来配置和初始化。可以使用应用服务器的管理控制台（作为JMX客户）来监视与管理MBean。

MBean可以打包为JMX服务，它不仅在支持JMX的应用服务器之间是可移植的而且对于运行中的系统它也是可配置的（热配置）。

Hibernate可以被打包并且被作为JMX MBean管理。Hibernate JMX服务允许Hibernate在应用服务器起动时初始化并且允许通过JMX客户进行控制（配置）。然而，JMX组件不能自动地

与容器管理的事务集成。因此，清单2.7（一个JBoss服务配置描述符）中列出的配置选项与通常Hibernate在管理环境中的设置看起来有些相似。

```
<server>
    <mbean
        code="net.sf.hibernate.jmx.HibernateService"
        name="jboss.jca:service=HibernateFactory, name=HibernateFactory">
        <depends>jboss.jca:service=RARDeployer</depends>
        <depends>jboss.jca:service=LocalTxCM,name=DataSource</depends>
        <attribute name="MapResources">
            auction/Item.hbm.xml, auction/Bid.hbm.xml
        </attribute>
        <attribute name="JndiName">
            java:/hibernate/HibernateFactory
        </attribute>
        <attribute name="Datasource">
            java:/comp/env/jdbc/AuctionDB
        </attribute>
        <attribute name="Dialect">
            net.sf.hibernate.dialect.PostgreSQLDialect
        </attribute>
        <attribute name="TransactionStrategy">
            net.sf.hibernate.transaction.JTATransactionFactory
        </attribute>
        <attribute name="TransactionManagerLookupStrategy">
            net.sf.hibernate.transaction.JBossTransactionManagerLookup
        </attribute>
        <attribute name="UserTransactionName">
            java:/UserTransaction
        </attribute>
    </mbean>
</server>
```

（清单2.7）

HibernateService依赖于其它两个JMX服务：service = RARDeployer和service =

LocalTxCM，name = DataSource，它们都位于jboss.jca服务域名里。

Hibernate MBean可以在包net.sf.hibernate.jmx里找到。很不幸，生命周期管理方法例如开始与停止JMX服务并不是JMX1.0规范的一部分。因此，HibernateService的start()与stop()方法是特定于JBoss应用服务器的。

注意 如果你对JMX的高级用法感兴趣，JBoss是一个很好的开源的起点：JBoss中所有的服务（甚至是EJB容器）都是作为MBean实现的并且都可以通过一个提供的控制台接口来管理。

我们推荐你在试图将Hibernate作为JMX服务运行之前，首先试验程序化地配置Hibernate（使用配置对象）。然而，许多特征（像Hibernate应用的热配置）一旦在Hibernate中可用了，可能也只能作为JMX使用。现在，Hibernate使用JMX的最大优点是自动起动；这意味着在你的应用代码中，你不再需要创建一个配置对象并构造一个会话工厂了，一旦HibernateService被配置并起动后你只需要通过JNDI简单地访问会话工厂就行了。

2．5 总结

本章在运行了一个简单的“Hello World”例子之后，我们在较高的层次上对Hibernate和它的体系结构进行了浏览。你也看到了如何在不同的环境中使用不同的技术配置Hibernate，甚至包括JMX。

接口Configuration（配置）和SessionFactory（会话工厂）是在管理与非管理两种环境中运行Hibernate应用的入口点。Hibernate还提供了另外的API，例如Transaction（事务）接口，来弥补这两种环境间的差异并允许你保持你的持续性代码的可移植性。

Hibernate可以被集成到几乎每一种Java环境中，可以是Servlet，Applet或者一个完全

管理的三层的客户/服务器应用。在Hibernate配置中最重要的元素是数据库资源（连接的配置），事务策略，当然还有基于XML的映射元数据。

Hibernate的配置接口被设计为包含尽可能多的使用场景同时还要易于理解。通常一个名为hibernate.cfg.xml的文件和一行代码就足够配置和运行Hibernate了。

如果没有持续类与XML映射文件的话这些东西都没有太多的使用价值。下一章我们将专注于编写和映射持续类。很快你就能够在一个包含重要的对象-关系映射的实际应用中存储与取出持续对象了。

Hibernate In Action

中文版

第四章 操作持久对象

4.1 持久化生命周期

由于 Hibernate 是透明的持久化机制—类不能意识到它们自己的持久能力—编写应用逻辑时不用意识到你所操纵的对象是持久状态还是存在于内存中的临时状态。当应用调用对象的方法时不需要关心它的状态是否是持久的。

然而，在持久状态的应用中，只要应用需要把内存中的状态传到数据库（反之亦然）就必须同持久层打交道。你可以调用 Hibernate 持久化管理器和查询接口来完成这种操作。当使用那种方式同持久层打交道时，应用关心与持久化相关的对象的状态及生命周期是必要的。我们将把它称为持久化生命周期。

对于持久化生命周期，不同的 ORM 实现使用不同的术语，定义不同的状态及状态转换。此外，内部使用的对象状态可能与其暴露给客户端应用的状态不同。Hibernate 仅仅定义了三种状态：瞬时、持久和分离，对客户端代码隐藏了其内部实现的复杂性。这一章，我们解释这三种状态：瞬时、持久和分离。

让我们在状态图中看看这些状态和它们的转换，如图 4.1 所示。你也可以看到调用持久管理器触发转换的方法。在这一节我们讨论这张图；以后你无论什么时候需要一个综述都可以引用它。

在其生命周期中，对象可以从瞬时对象转换到持久对象，再转换到分离对象。让我们仔细看看这些状态中的每一个状态。

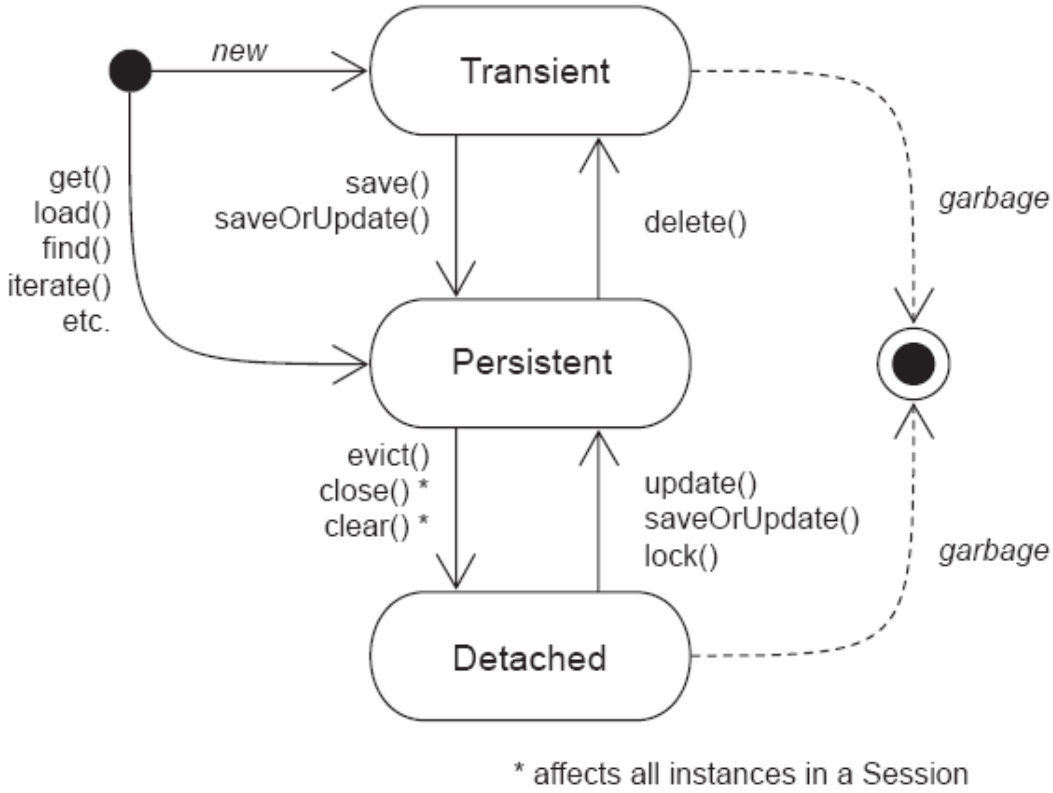

图 4.1Hibernate 中的对象状态及转换

4.1.1 瞬时对象

在 Hibernate 中，使用 new 操作符初始化的对象不是立刻就是持久的。它们的状态是瞬时的，也就是说它们没有跟任何数据库表的行相关联，只要应用不再引用这些对象（不再被任何其它对象所引用），它们的状态将会丢失。这时，那些对象的生命期将会有效地终止，变得不可访问，交给垃圾回收机制来回收。

Hibernate 认为所有的瞬时实例都是非事务的，瞬时实例状态的修改不能在任何事务的上下文中执行。这就意味着 Hibernate 不能对瞬时对象提供任何回滚功能。（实际上 Hibernate 不回滚任何对象所做的修改，以后你就会看到）。

默认地，仅仅被其他瞬时实例引用的对象也是瞬时的。把实例从瞬时状态转换为持久状态有两种方式：调用持久管理器的 save()方法或者从已经存在的持久实例中创建引用。

4.1.2 持久对象

持久实例是任何具有数据库标识的实例，就像第三章第 3.4 节“理解对象标识符”所定义的那样。也就是持久实例有一个主键值设为数据库标识符。

持久实例可能是由应用程序初始化的对象调用持久管理器（Hibernate Session，本章的后面部分将会详细讨论）的 save()方法实现持久化。然后，持久实例就跟持久管理器关联起来。它们甚至可能是引用另一个已经与持久管理器相关联的持久对象来实现持久化。可选的，持久实例可能是通过执行查询，通过标识符查找从数据库中检索出来的实例，或者是从另一个持久实例开始导航对象图。换句话说，持久实例通常是同 Session 相关的，是事务的。

持久实例是在事务中进行操作的—它们的状态在事务结束时同数据库进行同步。当事务提交时，通过执行 SQL 的 INSERT、UPDATE 和 DELETE 语句把内存中的状态同步到数据库中。这个过程也可能在其它时间发生。例如，Hibernate 在执行查询之前可能要同数据库同步。这就确保查询能够意识到在事务早期所做的更改。

我们称已经分配主键值但是还没有插入到数据库中的持久实例是新的。新的持久实例将会仍然保持“新”的状态直到同步发生。

当然，你不必在事务结束时把内存中的每个持久对象的状态更新到数据库中对应的行上。ORM 软件必须有一种机制来检测哪个持久对象已经被应用程序在事务中修改了。我们称其为自动脏数据检查（修改过的对象还没有同步到数据库中被任为是脏的）。这种状态对于应用程序来说是不可见的。我们把这种特性称为 *transparent transaction-level write-behind*，意思是 Hibernate 尽可能晚地把改变的状态同步到数据库中，但是对应用程序隐藏其实现细节。

Hibernate 能够确切地知道哪个属性改变了，因此可以在 SQL UPDATE 语句中仅包含那些需要更新的列。这样做可以提高性能，对于一些特定的数据库尤其如此。然而，这样做并不是总能得到性能提升，理论上，在某些环境中，可能要降低一些性能。因此，Hibernate 在 SQL UPDATE 语句中默认包含所有的列（因此，Hibernate 在启动时能够生成这个基本的 SQL，而不是在运行时）。如果仅仅希望更新修改过的列，你可以在类映射中通过把 dynamic-update 设

为 true 启动动态 SQL 生成功能。（注意，此特性在手工编码的持久层中是非常难以实现的。）下一章，我们详细讨论 Hibernate 的事务概念和同步过程（即 flushing）。

最后，持久实例通过调用持久管理器 API 的 delete()方法使其变成瞬时的，导致删除数据库中相应的行。

4.1.3 分离对象

当事务结束时，同持久管理器相关联的持久实例仍然存在（如果事务成功，它们在内存中的状态将会同数据库同步）。在具有过程范围标识（*process-scoped identity*，看下一节）的 ORM 解决方案中，这些实例保留着同持久管理器的关联并且仍然认为是持久的。

可是在 Hibernate 中，当你关闭 Session 时这些实例就失去了同持久管理器的关联。我们把这些对象称为分离的，表明这些状态不再跟数据库中的状态同步，不再在 Hibernate 的管理下。然而，它们仍然含有持久数据（可能是稳定的）。应用程序可能（通常）含有事务（及持久管理器）之外的分离对象的引用。Hibernate 可以让你同新的持久管理器重新关联这些实例以便在新事务中重用这些实例（重新关联后，这些对象被认为是持久的）。这种特性对怎样设计多层应用有很大影响。从一个事务中返回对象到表示层，以后在新的事务中重用它们的能力是 Hibernate 的一个主要卖点。我们在下一章讨论这种作为对于长时间运行的应用事务（*application transaction*）的一种实现技术的使用方法。我们也在第八章“重新考虑传输对象”这一节告诉你怎样通过使用分离的对象避免 DTO（anti-）模式。

Hibernate 也提供了一个显式的分离操作：Session 的 evict()方法。然而，这个方法只在 cache 管理中使用（考虑性能问题）。通常不显式地执行分离。然而，所有在事务中检索出来的对象在当 Session 关闭或当它们被序列化时（例如，它们被远程传递）就变成分离的。因此，Hibernate 不需要提供控制子图分离的功能。然而，应用程序能够使用查询语言或显式的图表导航来控制抓取子图（当前装载在内存中的实例）的深度。那么，当 Session 关闭时，整个子图（所有与持久管理器关联的对象）就会变成分离的。

让我们再来看看不同的状态，但是这次考虑对象标识的范围。

4.1.4 对象标识的范围

作为应用程序开发者，我们使用 Java 对象标识（a=b）来标识对象。因此，如果对象改变了状态，那么能够保证在新的状态中它的 Java 标识仍然相同么？在分层的应用程序中可能不会这样。

为了研究这个话题，有必要理解 Java 标识 a=b 与数据库标识 a.getId().equals(b.getId())之间的关系。有时它们是相等的，有时却不相等。我们把 Java 标识等于数据库标识的情况称为对象标识范围（scope of object identity）。

在这个范围内，通常有三种选择：

- 没有标识范围（*no identity scope*）的简单持久层不能保证如果某一行被访问两次，应用程序能够返回相同的 Java 对象实例。如果应用程序在一个单独的事务中修改了代表相同行的两个不同的实例将会出问题（你怎样决定哪一个状态将会同数据库同步？）。
- 使用事务范围标识（transaction-scoped identity）的持久层保证在单独的事务上下文中，仅仅有一个对象实例代表数据库中的某一行。这就避免了前面那个问题，并且允许做一些事务级的缓存。
- 过程范围标识（process-scoped identity）更进一步，它能保证在整个过程（JVM）中只有一个对象实例代表某一行。

对于典型的网络或企业应用程序，事务范围标识是首选的。过程范围标识在利用缓存和多个事务重用实例的编程模型方面有一些潜在的优点；然而在普遍的多线程应用程序中，同步共享访问全局标识图中的持久对象要花费很大代价。在每个事务范围内每个线程只同完全不同的一组持久实例工作将会更加简单、更易于升级。

宽松地讲，Hibernate 执行事务范围的标识。实际上，Hibernate 标识范围是 Session 的实例，如果对象在几个操作中使用相同的持久管理器（the Session）就能够保证这些对象是相等的。但是 Session 同（数据库）事务是不一样的—它是一个更复杂的元素。我们将会在下一章探索

这个概念的不同及结果。让我们再次关注持久化生命周期和标识范围。

如果你在同一个 Session 中使用相同的数据库标识符值请求两个对象，结果将会是对同一个内存对象的两个引用。下面的代码示例在两个 Session 中用几个 load()操作演示这种行为：

```
Session session1 = sessions.openSession();
Transaction tx1 = session1.beginTransaction();
// Load Category with identifier value "1234"
Object a = session1.load(Category.class, new Long(1234) );
Object b = session1.load(Category.class, new Long(1234) );
if ( a==b ) {
System.out.println("a and b are identical.");
}
tx1.commit();
session1.close();
Session session2 = sessions.openSession();
Transaction tx2 = session2.beginTransaction();
Object b2 = session2.load(Category.class, new Long(1234) );
if ( a!=b2 ) {
System.out.println("a and b2 are not identical.");
}
tx2.commit();
session2.close();
```

由于对象引用 a 和 b 在相同的 Session 中装载，它们不仅有相同的数据库标识，而且有相同的 Java 标识。然而，一旦超出了这种界限，Hibernate 就不能保证 Java 标识是相等的，因此，a 和 b2 是不相等的，信息打印在控制台上。当然，测试数据库标识--a.getId().equals (b2.getId())—将仍然返回 true。

为了更深入讨论标识范围，我们需要考虑持久层怎样持有对标识范围外的对象的引用。例如，对于像 Hibernate 这样的事务范围标识的持久层，对分离的对象（那就是在以前完成了的 Session 中持久或装载过的实例）进行引用是否是可容忍的？

4.1.5 标识范围之外

如果对象引用离开了能够得到保证的标识范围，我们把它称为对分离对象的引用。为什么这方面内容很有用呢？

在 web 应用程序中，通常不维护跨用户交互的数据库事务。用户花很长时间思考怎样修改，由于升级性的原因，你必须让数据库事务短而且尽可能快地释放数据库资源。在这种环境中，能够重用对分离实例的引用非常有用。例如，你可能希望把在一个工作单元中检索的对象传给表示层，以后，用户修改过它之后在第二个工作单元重用它。

通常你不希望在第二个工作单元重新绑定整个对象图，由于性能（或其它）原因，有选择地重新关联分离的实例很重要。Hibernate 支持选择性地重新关联分离实例（*selective reassociation of detached instances*）。这意味着应用程序能有效地将分离对象图的子图重新绑定到当前（或第二个）Hibernate Session。一旦分离对象重新绑定到新的 Hibernate 持久管理器，这个对象就会被认为是持久实例，它的状态将会在事务结束时同数据库同步（由于 Hibernate 自动检查持久实例的脏数据）。

当创建从分离实例到新的瞬时实例的引用时，重新绑定可能会导致在数据库中创建新的行。例如，新的 Bid 在表示层的时候就可能已经加到分离的 Item。Hibernate 能够察觉到 Bid 是新的，必须插入到数据库中。为了使其工作，Hibernate 必须能够分辨“新”的瞬时实例和“老”的分离实例。瞬时实例（如 Bid）可能需要保存；分离实例（如 Item）可能需要重新绑定（以后会在数据库中更新）。有几种方式区分瞬时实例和分离实例，但是最好的方式是查看标识符属性值。Hibernate 能在重新绑定时检查瞬时对象或分离对象的标识符并能正确处理对象（和关联的对象图）。我们将在第 4.3.4 节“区分瞬时和持久实例”详细讨论这个重要内容。

如果你想要在你自己的应用程序中利用 Hibernate 对重新关联附加实例的支持，你需要在设计应用程序时知道 Hibernate 的标识范围—那就是保证标识实例的 Session 范围。只要离开了那个范围就会有分离实例，就会出现另一个有趣的概念。

我们需要讨论 Java 相等（看第三章第 3.4.1 节“标识和相等”）和数据库标识之间的关系。

相等是一个标识概念，你作为类开发者，可以（有时不得不）控制和使用具有分离实例的类。Java 相等通过在业务模型的持久类中实现 equals()和 hashCode()来定义。

4.1.6 实现 equals()和 hashCode()

equals()方法被应用程序代码调用，更重要的是被 Java 集合调用。例如，Set 集合，把每个对象放进此集合中都会调用 equals()方法来确定（阻止）重复的元素。

首先，让我们考虑由 java.lang.Object 定义的默认的 equals()实现，它是通过 Java 标识进行比较的。Hibernate 为 Session 中数据库的每一行分配一个唯一的实例。因此，如果你从不混合实例，默认的 equals()方法是正确的—那就是，你从不把不同 session 中的分离对象放进相同的 Set。（实际上，如果分离的对象来自相同的 session 但是在不同的区域序列化和反序列化，我们探索的这个话题仍然适用）。然而，只要你有来自多个 session 的实例，就有可能在一个 Set 中包含两个 Item，每个 Item 代表数据库表中相同的行但是没有相同的 Java 标识。这可能被认为是语法错误。然而，只要你按照原则处理不同 session 的分离对象（也要关注序列化和反序列化），建立标识（默认的）相等的复杂应用程序是可能的。这种方法的好处是不用写额外的代码来实现相等。

然而，如果这种相等的内容不是你想要的，你不得不在持久类中重写 equals()方法。记住，当你重写 equals()方法时也要重写 hashCode()方法以便两种方法保持一致（如果两个对象是相等的，他们必须有相同的 hashcode）。让我们看看在持久类中重写 equals()和 hashCode()方法的几种方式。

使用数据库标识符相等

比较聪明的方式是仅仅比较数据库标识符属性（通常是代理主键）值来实现 equals()方法：

```
public class User {
...
public boolean equals(Object other) {
if (this==other) return true;
if (id==null) return false;
if ( !(other instanceof User) ) return false;
```

```
final User that = (User) other;
return this.id.equals( that.getId() );
}
public int hashCode() {
return id==null ?
System.identityHashCode(this) :
id.hashCode();
}
}
```

注意这个 equals()方法对还没有分配数据库标识符值的瞬时实例（如果 id==null）是怎样转而求助 Java 标识符的。这是合理的，因为它们没有同另一个实例相同的持久标识。

不幸的是这种方案有个很大的问题：Hibernate 直到实体被保存了之后才分配标识符值。因此，如果对象在被保存之前加到 Set 中，当对象被包含到 Set 中时 hashcode 也改变了，跟 java.util.Set 的内容正好相反。特别地，这就使集合的级联保存（在本章的后面部分讨论）变得毫无用处。我们不赞同这种解决方案（数据库标识符相等）。

值比较

较好的方法是在 equals()比较中除了包含数据库标识符属性之外还要包含持久类的所有其它持久属性。这就是多数人怎样理解 equals()含义的，我们把它称为值相等。

我们所说的“所有属性”不包括集合。集合的状态与不同的表有关，所以包含它是错误的。更重要的是，你不想仅仅为了执行 equals()而强迫检索整个对象图。在 User 那个例子中，这意味着你在比较时不必包含 items 集合（此用户售出的项目）。因此，下面才是你应该使用的实现代码：

```
public class User {
...
public boolean equals(Object other) {
if (this==other) return true;
if ( !(other instanceof User) ) return false;
final User that = (User) other;
if ( !this.getUsername().equals( that.getUsername() )
```

```
return false;
if ( !this.getPassword().equals( that.getPassword() )
return false;
return true;
}
public int hashCode() {
int result = 14;
result = 29 * result + getUsername().hashCode();
result = 29 * result + getPassword().hashCode();
return result;
}
}
```

然而，这种方法也有两个问题：

- 来自不同 session 的实例如果其中的一个修改了（例如，如果用户修改了密码），它们将不再相等。
- 具有不同数据库标识的实例（代表数据库表中不同行的实例）被认为是相等的，除非有一些组合属性被赋成唯一的（数据库列有唯一性约束）。在 User 例子中有一个唯一性属性：username。

为了开始我们推荐的解决方案，你需要理解业务键的概念。

使用业务键相等

业务键是一个属性，或是一些属性的组合，对于具有相同数据库标识的每个实例是唯一的。本质上，如果不使用代理键，选择业务键是很正常的。不像自然主键，对于业务键从不改变的实例来说它不是必须的—只要业务键改变，那就需要。

我们要求每个实体都应该有业务键，即使包含了类的所有属性（这对一些不可变类来说是正确的）。用户认为业务键唯一地标识某条记录，不管应用程序和数据库使用什么样的代理键。

业务键相等意味着 equals()方法仅仅比较形成业务键的属性。这就是避免前面提到的所有问题的最好解决方案。唯一不好的是首先需要额外定义正确的业务键。但是这些付出是必要

的，如果想让数据库通过约束检查来确保数据完整性，定义唯一性键非常重要。

对于 User 类，username 是非常好的候选业务键。username 从不为 null，唯一而且很少改变：

```
public class User {
...
public boolean equals(Object other) {
if (this==other) return true;
if ( !(other instanceof User) ) return false;
final User that = (User) other;
return this.username.equals( that.getUsername() );
}
public int hashCode() {
return username.hashCode();
}
}
```

其它类业务键可能更复杂，由复合属性组成。例如，Bid 类的候选业务键是项目 ID 与投标量的组合或项目 ID 同投标日期和时间的组合。BillingDetails 抽象类的好的业务键是 number 同帐单细目种类（子类）的组合。注意，在子类中重写 equals()并且在比较中包含另一个属性是不对的。在本例中，既满足对称性的又满足可传递性的相等需求是狡猾的。更重要的是业务键不能同任何数据库中定义好的候选自然键相符（子类属性可能映射到不同的表）。

你可能已经注意到 equals()和 hashCode()方法一直通过 getter 方法访问另一个对象的属性。这很重要，因为以 other 传递的对象实例可能是代理对象，而不是持有持久状态的真正实例。这一点是 Hibernate 不是完全透明的原因之一，但是使用访问器方法而不是访问直接的实例变量是最好的实践。

最后，注意当修改业务键属性的值时，当业务对象在集合中不要改变其值。

本节我们已经讨论过持久管理器。是仔细看看持久管理器及详细地探索 Hibernate Session API 的时候了。我们将在下一章重新详细讨论分离对象。

4.2 持久管理器

任何透明的持久管理器都含有持久管理器 API，通常提供下列服务：

- 基本的 CURD 操作；
- 执行查询；
- 控制事务；
- 事务级的缓存管理；

持久管理器可以暴露给几个不同的接口（就 Hibernate 来说，Session，Query，Criteria 和 Transaction）。这些接口的内部实现连接的很紧密。

应用程序与 Hibernate 之间的核心接口是 Session，它是刚刚列出的所有操作的起点。我们将在这本书余下的大部分章节交替谈到 persistence manager 和 session，这是同 Hibernate 社区中的用法一致的。

那么，怎样开始使用 session 呢？在工作单元的开头，一个线程含有从应用程序的 SessionFactory 返回的 Session 实例。应用程序如果访问多个数据源可能含有多个 SessionFactory。但是仅仅为了服务于某个请求你不要创建一个新的 SessionFactory—创建 SessionFactory 需要耗费大量的资源。另一方面，创建 Session 耗费的资源却很少。Session 甚至只有需要连接时才获得 JDBC 连接。

打开新的 Session 之后，就可以用它装载和保存对象了。

4.2.1 使对象持久化

想用 Session 做的第一件事就是把新创建的瞬时对象持久化。用 save()方法来做这件事情：

```
User user = new User();
user.getName().setFirstname("John");
user.getName().setLastname("Doe");
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
session.save(user);
tx.commit();
```

```
session.close();
```

首先，我们同往常一样初始化一个新的瞬时对象 user。当然，也可以在打开 session 之后初始化它，它们仍然是不相关的。我们使用 SessionFactory 打开一个新的 Session，然后开始一个新的数据库事务。

调用 save()方法使 User 的瞬时实例持久化。现在就跟当前 session 关联起来。然而，还没有执行 SQL 的 INSERT 语句。Hibernate Session 只有完全必要时才执行 SQL 语句。

对持久实例做的更改必须在某一时刻同数据库同步，这在 commit() Hibernate 事务时发生。在这个例子中，Hibernate 获得 JDBC 连接，然后执行 SQL INSERT 语句。最后，Session 关闭，释放 JDBC 连接。

注意在同 Session 关联之前最好（但不是必须的）完全初始化 User 实例。SQL 的 INSERT 语句含有对象调用 save()时持有的值。当然可以在调用完 save()之后修改对象，并且你所做的更改将会通过 SQL 的 UPDATE 语句同数据库同步。

在 session.beginTransaction()和 tx.commit()之间发生的任何事情都在一个数据库事务中。我们还没有详细地讨论过事务，我们将在下一章讨论。但是记住一个事务范围内的所有数据库操作或者完全成功或者完全失败。如果一条 INSERT 或 UPDATE 语句在 tx.commit()时失败，在这个事务中对持久对象所做的所有改变将会回滚到数据库级别。然而，Hibernate 不会回滚在内存中对持久对象所做的更改。这是合理的，因为数据库事务的失败通常是不可恢复的，你不得不立即丢弃失败的 Session。

4.2.2 更新分离实例的持久状态

在 session 关闭之后修改 user 将不会对数据库的持久表示层有影响。当 session 关闭时，user 变成一个分离实例。它可以在新的 session 中通过调用 update()或 lock()重新关联。

update()方法通过调度 SQL 的 UPDATE 语句强制更新数据库中对象的持久状态。下面有一个操作分离对象的例子：

```
user.setPassword("secret");
Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();
sessionTwo.update(user);
user.setUsername("jonny");
tx.commit();
sessionTwo.close();
```

对象是否在被传给 update()之前或之后修改过了都没有关系。重要的是调用 update()方法来重新将分离实例同新的 Session（当前事务）关联并且告诉 Hibernate 把此对象当作脏数据来对待（除非在持久类映射中设置了 select-before-update，这种情况下，Hibernate 就会通过执行一个 SELECT 语句把对象的当前状态同当前数据库的状态相比较来决定对象是否是脏的）。

调用 lock()方法将对象同 Session 相关联而不强制更新，就像下面这样：

```
Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();
sessionTwo.lock(user, LockMode.NONE);
user.setPassword("secret");
user.setLoginName("jonny");
tx.commit();
sessionTwo.close();
```

在这个例子中，是否在对象同 session 相关联之前或之后做了更改很重要。在调用 lock()之前做的更改不会同步到数据库中，只有你能够确信分离对象没有做过更改才能使用 lock()。

我们在下一章讨论 Hibernate 锁定模式。在这里我们通过指定 LockMode.NONE 告诉 Hibernate 当重新将对象同 Session 相关联时不执行版本检查或获得任何数据库级别锁。如果我们指定了 LockMode.READ 或 LockMode.UPDATE，Hibernate 将会执行 SELECT 语句来进行版本检查（并且设置更新锁）。

4.2.3 检索持久对象

Session 也用来查询数据库，检索存在的持久对象。Hibernate 在这个领域尤为强大，你将在本章的后面和第七章看到。然而，Session API 为最简单的查询提供了特定的方法：通过标

识符检索。这些方法其中之一是 get()，如下所示：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
tx.commit();
session.close();
```

检索出来的对象 user 现在可能被传递给表示层在事务之外作为分离对象（session 关闭之后）使用。如果数据库中没有给定标识符值对应的行，get()方法返回 null。

4.2.4 更新持久对象

任何通过 get()或其它种类的查询返回的持久对象已经同当前 Session 和事务上下文相关联了。可以修改对象并把修改后的状态同步到数据库中去。这种机制叫做自动脏数据检查，那意味着 Hibernate 将会跟踪和保存在一个 session 中对对象所做的更改。

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
user.setPassword("secret");
tx.commit();
session.close();
```

首先我们需要根据给定的标识符从数据库中检索对象。修改对象，然后当调用 tx.commit()时把这些修改同步到数据库中去。当然，一旦我们关闭了 Session，这个实例就被认为是分离的。

4.2.5 把持久对象转换为瞬时的

将持久对象转为瞬时对象很容易，使用 delete()方法将它的持久状态删除：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
```

```
session.delete(user);
tx.commit();
session.close();
```

只有在事务结束 Session 同数据库同步时才执行 SQL 的 DELETE 语句。

Session 关闭以后，user 被认为是普通的瞬时实例。瞬时对象如果不再被其它对象引用将由垃圾回收器销毁。无论是内存中的对象实例还是持久的数据库行都将被移除。

4.2.6 把分离对象转换为瞬时的

最后，你可以将分离的实例转为瞬时的，从数据库中删除它的持久状态。这意味着不必重新绑定分离的实例就可以从数据库中删除它，可以直接删除分离的实例：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
session.delete(user);
tx.commit();
session.close();
```

在这个例子中，调用 delete()做两件事：将对象同 Session 关联然后为删除准备对象，在 tx.commit()时执行。

现在你已经知道持久生命期和持久管理器的基本操作。同第三章讨论的持久类映射一起，你可以创建自己的小型 Hibernate 应用程序了。（如果你喜欢，可以跳到第八章阅读 SessionFactory 和 Session 管理器的 Hibernate 帮助类。）记住我们没有给你任何异常处理代码，但是你应该能够自己解决 try/catch 块。映射一些简单的实体类和组件，然后在单独的应用程序中存储和装载对象（你不需要 web 容器或应用服务器，仅仅写一个 main 方法）。然而，只有你存储关联的实体对象—那就是，当你处理更复杂的对象图时—你就会知道在每个对象图中调用 save()或 delete()不是写应用程序的有效方式。

你最好尽可能少地调用 Session。传递的持久化（Transitive persistence）对强迫改变对象状态及控制持久化生命周期提供了更自然的方式。

4.3 在 Hibernate 中使用传递的持久化

实际上，重要的应用程序不是操纵单个对象而是操作对象图。当应用程序操纵持久对象图时，结果可能是由持久、分离和瞬时实例组成的对象图。传递的持久化是一种允许你把持久化自动传递给瞬时和分离子图的技术。

例如，如果我们为已经持久化的分层类添加一个新初始化的 Category，那么它不用调用 Session.save()就能自动持久化。在第三章我们映射 Bid 和 Item 之间的父子关系时给出了一个有点不同的例子。在那个例子中，不仅当 bid 加到 item 时会自动持久化，而且当拥有它们的 item 删除时也会自动地被删除。

传递的持久化模型不只一个。最有名的是可到达性的持久化（persistence by reachability），我们首先讨论它。虽然一些基本的原则是相同的，Hibernate 使用它自己的更强大的模型，以后你就会看到。

4.3.1 可到达性的持久化

如果当应用程序从另一个已经持久化的实例创建对实例的对象引用时所有实例都变成持久的，这样的对象持久层就称为实现了可到达性持久化。这种行为如图 4.2 的对象图（注意不是类图）所示：

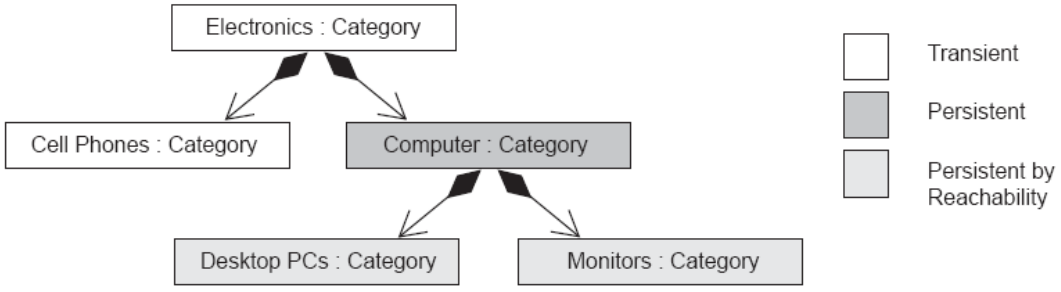

Figure 4.2 Persistence by reachability with a root persistent object

在这个例子中，“Computer”是持久的对象。对象“Desktop PCs”和“Monitors”也是持久的，它们从“Computer”Category 实例可以到达。“Electronic”和“Cell Phones”是瞬时的。注意我们假设导航仅仅对子目录是可以的，对父目录不可以—例如，我们可以调用

computer.getChildCategories()。可到达性的持久化是递归的算法：从一个持久实例开始所有可到达的对象在两种时候变成持久的，或者当原始对象持久化时或者当内存状态同数据存储同步前。

可到达性的持久化保证完整性约束，任何对象图能够通过装载持久的根对象完全重建。应用程序可以遍历对象图，从一个关联到另一个关联而不用担心实例的持久状态。（SQL 数据库使用不同的方法保证完整性约束，依靠外键和其它约束来检测越界（misbehaving）的应用程序。

在可到达性的持久化形式中，数据库有一些顶级的（或根的）对象，从这些对象可以到达所有的持久对象。实际上，如果一个实例不能通过根持久对象引用到达，这个实例就应该变成瞬时的并从数据库中删除。

Hibernate 和其它 ORM 方案都不能实现这种形式，在 SQL 数据库中没有类似于根持久对象的东西，也没有能够检测未引用实例的持久垃圾收集器。面向对象的数据存储实现的垃圾收集算法可能跟 JVM 的内存对象实现的算法相似，但是这种方式不适用于 ORM 世界，为了未引用的列而检索所有的表是不现实的。

因此，可到达性的持久化是最好的折中解决方案。它帮你把瞬时实例持久化并把他们的状态同步到数据库中而不用多次调用持久管理器。但是（至少在 SQL 上下文和 ORM 中）它不是解决把持久对象转为瞬时和从数据库中删除它们状态的问题的完全解决方案。这是更复杂的问题。当你删除某个对象时不能简单的删除所有可到达的对象，其它持久实例可能持有对它们的引用（记住实体是可以共享的）。你甚至不能安全地删除那些内存中不被任何持久对象引用的实例，内存中的实例仅仅是表示数据库的所有对象的一个小的子集。让我们看看 Hibernate 更复杂的传递持久化模型。

4.3.2 Hibernate 的级联持久化

Hibernate 的传递持久化模型使用跟可到达性持久化相同的基本内容—那就是检查对象关系来决定传递状态。然而，Hibernate 允许你为每个关系映射指定级联形式，为所有的状态转

换提供更复杂更精确的控制。Hibernate 读取声明的状态并自动级联操作到相关的对象。

当查找瞬时或分离对象时，Hibernate 默认不导航关联，因此保存、删除或重新绑定 Category 不会影响子目录对象。这跟可到达性的持久化的默认行为正好相反。如果对于某个关联想使用传递的持久化，你必须在映射元数据中重写这个默认的行为。

你可以在元数据中用下面的属性映射实体关系：

- cascade="none"，默认值，告诉 Hibernate 忽略关系。
- cascade="save-update"告诉 Hibernate 在下面这些情况导航关联：当事务提交时，当对象传给 save()或 update()方法并保存新初始化的瞬时实例及把更改持久到分离实例时。
- cascade="delete"告诉 Hibernate 当对象传给 delete()时导航关联并删除持久实例。
- cascade="all"意思是 save-update 和 delete 都级联，就像调用 evict 和 lock。
- cascade="all-delete-orphan"，跟 cascade="all"一样，但是除此之外，Hibernate 删除任何已经从关联（例如，从集合）删除（不再被引用）的持久实体实例。
- cascade="delete-orphan"，Hibernate 将会删除任何已经从关联（例如，从集合）删除（不再被引用）的持久实体实例。

这种关联级的级联形式模型比可到达性的持久化丰富但是缺少安全性。Hibernate 不能提供与可到达性持久化提供的相同的引用完整性。相反，Hibernate 部分代理与默认关系数据库的外键约束有关的引用完整性。当然，这种设计决定有一个好的理由：它允许 Hibernate 应用程序有效地使用分离对象，因为可以在关联级上控制分离对象图的重新绑定。

让我们用一些关联映射的例子详细阐述级联的内容。我们推荐你通读下一节，因为每个例子是建立在以前的例子之上的。我们的第一个例子很简单，有效地保存新增加的目录。

4.3.3 管理拍卖类别

系统管理员可以在目录树中创建新目录，重命名目录及删除目录。这种结构如图 4.3 所示。

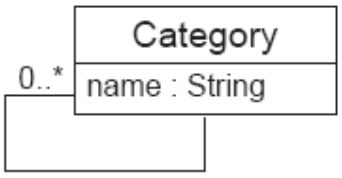

Figure 4.3
Category class with association to itself

现在，我们映射这个类和关系：

```
<class name="Category" table="CATEGORY">
...
<property name="name" column="CATEGORY_NAME"/>
<many-to-one
name="parentCategory"
class="Category"
column="PARENT_CATEGORY_ID"
cascade="none"/>
<set
name="childCategories"
table="CATEGORY"
cascade="save-update"
inverse="true">
<key column="PARENT_CATEGORY_ID"/>
<one-to-many class="Category"/>
</set>
...
</class>
```

这是递归双向的一对多关系，就像第三章简单讨论的那样。“一”的这一端映射 <many-to-one> 元素和用 <set> 映射 Set 类型的属性。两者都引用相同的外键列：PARENT_CATEGORY_ID。

假设我们创建一个新的 Category 作为“Computer”的子目录（如图 4.4）。

我们有几种方式创建这个新的“Laptops”对象并把它保存到数据库中。我们回到数据库

中检索新的“Laptops”目录属于的“Computer”目录，增加新的目录并提交事务：

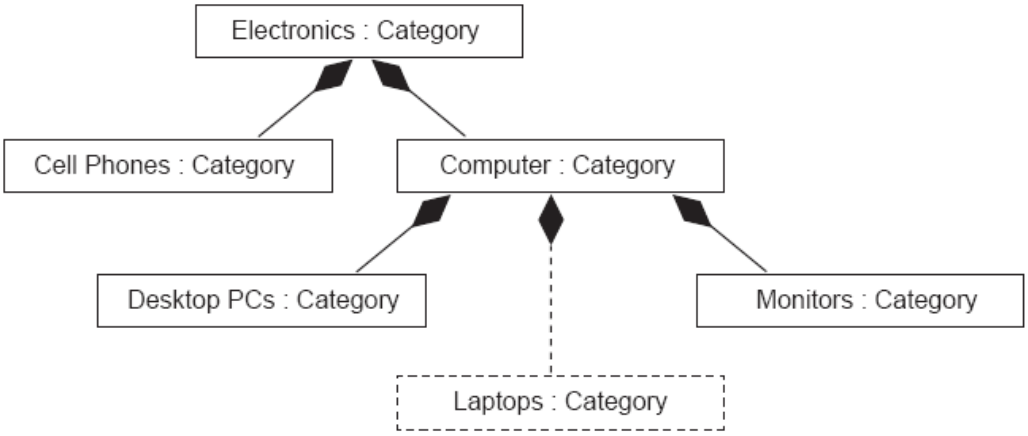

Figure 4.4 Adding a new `Category` to the object graph

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
Category computer = (Category) session.get(Category.class, computerId);
Category laptops = new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
tx.commit();
session.close();
```

computer 实例是持久的（绑定到 session），childCategory 关联已经打开了级联保存。因此，这段代码导致调用 tx.commit()时新的 laptops 目录成为持久的，因为 Hibernate 级联脏数据检查到 computer 的儿子。Hibernate 执行 INSERT 语句。

让我们再一次作相同的事，但是这次在任何事物之外（真正的应用程序中，在表示层操纵对象图很有用—例如，在把图传回持久层之前改变持久状态）创建“Computer”和“Laptops”之间的连接：

```
Category computer = ... // Loaded in a previous session
Category laptops = new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
```

现在分离的 computer 对象及它引用的其它分离对象与新的瞬时 laptops 对象相关联（反之亦然）。我们在第二个 Hibernate session 中保存新的对象把更改持久化到对象图中：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Persist one new category and the link to its parent category
session.save(laptops);
tx.commit();
session.close();
```

Hibernate 将会检查 laptops 父目录的数据库标识符属性，在数据库中正确创建与“Computer”的关系。Hibernate 把父亲的标识符值插入到 CATEGORY 中新的“Laptops”行的外键域。

由于 cascade=”none”是为 parentCategory 关联定义的，Hibernate 忽略层次树（“Computer”，“Electronics”）中任何其它目录的改变。调用 save()方法不会级联到被这种关联引用的实体。如果我们在映射 parentCategory 的<many-to-one>中设置 cascade=”save-update”，Hibernate 将会不得不导航内存中的整个对象图，将所有的实例同数据库同步。这个过程执行起来很糟糕，因为需要很多无用的数据访问。这种情况，我们对 parentCategory 既不需要也不想使用传递的持久化。

为什么有级联操纵？我们可以像前一个例子那样保存 laptop 对象，不使用任何级联映射。那么，考虑下面的例子：

```
Category computer = ... // Loaded in a previous Session
Category laptops = new Category("Laptops");
Category laptopAccessories = new Category("Laptop Accessories");
Category laptopTabletPCs = new Category("Tablet PCs")
laptops.addChildCategory(laptopAccessories);
laptops.addChildCategory(laptopTabletPCs);
computer.addChildCategory(laptops);
```

（注意，我们使用方便的 addChildCategory()方法在一端调用时设置关联连接的两端，就像第三章所描述的那样。）

不得不分别保存三个新的目录是我们所不希望的。幸运的是，因为我们把 childCategory 关联映射为 cascade=”save-update”，我们就不必那么做。我们用与以前相同的代码在新的 session 中保存单个“Laptops”目录就会保存所有三个新的目录了：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Persist all three new Category instances
session.save(laptops);
tx.commit();
session.close();
```

你可能想知道为什么级联形式叫做 cascade=”save-update”而不是 cascade=”save”。我们刚刚在前面把三个目录都持久化了，假设我们在接下来的请求中（session 和事务之外）对目录层次树作一些改变：

```
laptops.setName("Laptop Computers");
laptopAccessories.setName("Accessories & Parts");
laptopTabletPCs.setName("Tablet Computers");
Category laptopBags = new Category("Laptop Bags");
laptops.addChildCategory(laptopBags);
```

我们增加了一个新的目录作为“Laptops”目录的孩子并修改了三个已经存在的目录。下面的代码把这些改变同步到数据库中：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Update three old Category instances and insert the new one
session.update(laptops);
tx.commit();
session.close();
```

在 childCategories 关联中指定 cascade=”save-update”准确地反映了 Hibernate 能够决定需要什么把对象持久化到数据库中。这种情况，Hibernate 将会重新绑定或更新那三个分离的目录（laptops，laptopAccessories 和 laptopTabletPCs）并保存新的子目录（laptopBags）。

注意最后一个代码示例与前面两个在单独方法中调用的 session 示例不同。最后一个示例

使用 update()而不是 save()，因为 laptops 已经持久化了。

我们可以用 saveOrUpdate()方法重写所有的例子。然后那三个代码片断就会一样了：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Let Hibernate decide what's new and what's detached
session.saveOrUpdate(laptops);
tx.commit();
session.close();
```

saveOrUpdate()方法告诉 Hibernate 如果实例是新的瞬时实例就创建新的数据库行把实例的状态同步到数据库中，如果实例是分离的实例就更新已经存在的行。换句话说，它对 laptops 目录做的事情跟 cascade=”save-update”对 laptops 的子目录做的事情一样。

最后一个问题：Hibernate 怎样知道哪个孩子是分离的实例哪个是新的瞬时实例？

4.3.4 区分瞬时实例和分离实例

由于 Hibernate 不保留对分离实例的引用，你不得不让 Hibernate 知道怎样区分像 laptops（如果它在前一个 session 中创建）的分离实例和像 laptopBags 的新的瞬时实例。

可以利用一些选项。Hibernate 会把一个实例认为是未保存的瞬时实例，如果：

- 标识符属性（如果存在）为 null。
- 版本属性（如果存在）为 null。
- 在类的映射文档中支持 unsaved-value 及标识符属性匹配的值。
- 在类的映射文档中支持 unsaved-value 及版本属性匹配的值。
- 支持 Hibernate Interceptor 并在代码中检查完实例后从 Interceptor.isUnsaved()返回 Boolean.TRUE。

在我们的业务模型中，已经到处使用过可空的类型 java.lang.Long 作为标识符属性类型。由于我们正在使用生成的复合标识符，这就可以解决问题。新的实例有空的标识符属性值，因此 Hibernate 认为它们是瞬时的。分离的实例有非空的标识符值，Hibernate 也会正确的对待它们。

然而，如果在持久类中使用原始的 long 类型，就需要在所有的类中使用下面的标识符映射：

```
<class name="Category" table="CATEGORY">
<id name="id" unsaved-value="0">
<generator class="native"/>
</id>
....
</class>
```

unsaved-value 属性告诉 Hibernate 把具有标识符值为 0 的 Category 实例当作新初始化的瞬时实例。unsaved-value 属性的默认值为 null，因此，由于我们选择 Long 为标识符属性类型，可以在我们的拍卖应用程序类中忽略 unsaved-value 属性（我们到处使用相同的标识符类型）。

未保存的分配标识符	这种方式很适合复合标识符，但是不适合程序分配的键，包括遗留系统中的联合键。我们将在第八章第 8.3.1 节“遗留模式和联合键”讨论这个问题。在新的应用程序中尽量避免使用程序分配的键。

如果需要保存和删除对象，现在就能优化 Hibernate 应用程序和减少调用持久管理的次数了。检查所有类的 unsaved-value 属性，对分离对象做实验以便获得关于 Hibernate 传递的持久模型的感性知识。

现在我们看看另外一个重要的内容：怎样得到数据库之外的持久对象图（那就是怎样装载对象）。

4.4 检索对象

从数据库中检索对象是使用 Hibernate 最有趣（也是最复杂）的部分。Hibernate 提供下列方式从数据库中提取对象：

- 导航对象图，从一个已经装载的对象开始，通过像 aUser.getAddress().getCity()的属性访问器方法访问相关的对象。如果 Session 是打开的，当你导航图时，Hibernate 会自动装载图的节点。

- 当对象的唯一标识符值是已知的时候，通过标识符检索是最方便最有性能的方法。
- 使用 Hibernate 查询语言（HQL），它是完全面向对象的查询语言。
- 使用 Hibernate 条件 API，它提供了类型安全的面向对象的方式执行查询而不需要操纵字符串。这种便利性包括基于例子对象的查询。
- 使用本地 SQL 查询，这种查询 Hibernate 只关心把 JDBC 结果集映射到持久对象图。

在 Hibernate 应用程序中，将结合使用这几种技术。每一种检索方法可能使用不同的抓取策略—那就是定义持久对象图的哪个部分应该检索的策略。目标是在你的应用程序中为每个使用场合发现最好的检索方法和抓取策略，同时最小化查询语句的数量以获得最好的性能。

在这一节我们不仔细讨论每个检索方法，相反，我们将集中于基本的抓取策略和怎样调整 Hibernate 映射文件以便对所有的方法达到最好的默认抓取性能。在看抓取策略之前，我们将给出检索方法的概述。（我们提到 Hibernate 缓存系统但是将在下一章完全研究它。）

让我们开始最简单的例子，通过给定的标识符值检索对象（导航对象图不加以说明了）。在这一章的前半部分你已经看过一个简单的通过标识符检索的例子，但是还有许多需要知道。

4.4.1 根据标识符检索对象

下面的 Hibernate 代码片断从数据库中检索 User 对象：

```
User user = (User) session.get(User.class, userID);
```

get()方法很特别，因为标识符唯一地标识类的单个实例。因此，应用程序通常使用标识符方便地处理持久对象。当用标识符检索对象时可以使用缓存，如果对象已经缓存了可以避免数据库碰撞（hit）。

Hibernate 也提供了 load()方法：

```
User user = (User) session.load(User.class, userID);
```

load()方法是旧的，因为用户的请求已经把 get()方法加入到 Hibernate API。不同之处微不足道：

- 如果 load()方法不能在缓存或数据库中找到对象会抛出异常。load()方法从不返回

null。如果对象没找到 get()方法返回 null。

- load()方法返回代理而不是真正的持久实例。代理是一个占位符，当第一次调用它时才装载真正的对象。我们将在本节的后半部分讨论代理。另一方面，get()方法从不返回代理。

在 get()和 load()之间选择很简单：如果你能确定持久实例存在，不存在将会认为是异常，那么 load()是很好的选择。如果你不能确定给定的标识符是否有一个实例，使用 get()测试返回值，看它是否为 null。使用 load()有更深的含义：应用程序可能检索一个对持久实例的引用（代理）而不会强制数据库检索它的持久状态。因此，在缓存或数据库中不能找到持久对象时 load()不能抛出异常。异常会在以后抛出，当代理被访问的时候。

当然，使用标识符检索对象没有使用任意的查询复杂。

4.4.2 介绍 HQL

Hibernate 查询语言是与其相似的关系型查询语言 SQL 的面向对象方言。HQL 与 ODMG OQL 和 EJB-QL 很相像，但是不像 OQL，它是用于 SQL 数据库的，并且比 EJB-QL 更强大更优秀（然而，EJB-QL3.0 将会与 HQL 非常相似。）只要有 SQL 基础 HQL 非常容易学。

HQL 不是像 SQL 这样的数据操纵语言。它只能用来检索对象，不能更新、插入或删除数据。对象状态同步是持久管理器的工作，而不是开发者的工作。

大部分时间你仅仅需要检索特定类的对象，并且受那个类的属性的约束。例如，下面的查询根据姓来检索用户：

```
Query q = session.createQuery("from User u where u.firstname = :fname");
q.setString("fname", "Max");
List result = q.list();
```

准备查询 q 之后，我们把标识符值绑定到命名参数 fname 上。User 对象的 List 作为结果返回。

HQL 功能非常强大，虽然你不会一直使用其高级特征，但是你将会需要它们来解决一些复杂问题。例如，HQL 支持下面这些功能：

- 通过引用或持有集合（使用查询语言导航对象图）把限制条件应用到相关的关联对象的属性上的能力。
- 在事务范围仅仅检索一个或多个实体的属性而不是装载整个实体的能力。有时把它称为 *report query*，更正确的说法是 *projection*。
- 排列查询结果的能力。
- 分页查询的能力。
- 使用 group、having 及统计函数（如 sum、min 和 max）进行统计。
- 当在一行中检索多个对象时使用外联接。
- 调用用户自定义的 SQL 函数的能力。
- 子查询（嵌套查询）。

我们将在第七章把所有这些特性同可选的本地 SQL 查询机制放到一起讨论。

4.4.3 通过条件查询（Query by Criteria）

Hibernate 的通过条件查询（*query by criteria*(QBC)）API 允许你在运行时通过操纵查询对象来建立查询。这种方法允许动态的指定约束而不是直接操纵字符串，但是，它也丢掉了许多 HQL 的复杂性或强大功能。另一方面，以条件表示的查询比以 HQL 表示的查询可读性差。

通过名字检索用户使用查询对象更简单：

```
Criteria criteria = session.createCriteria(User.class);
criteria.add( Expression.like("firstname", "Max") );
List result = criteria.list();
```

Criteria 是一个 Criterion 实例树。Expression 类提供返回 Criterion 实例的静态工厂方法。一旦建立了想要的查询树，就会对数据库执行。

许多开发者喜欢 QBC，把它认为是更复杂的面向对象方法。他们也喜欢查询语法在编译时解释和验证的事实，而 HQL 只有在运行时才解释。

关于 Hibernate Criteria API 最好的方面是 Criterion 框架。这个框架允许用户对其进行扩展，像 HQL 这样的查询语言却很困难。

4.4.4 通过例子查询（Query by example）

作为 QBC 便利性的一部分，Hibernate 支持通过例子查询（QBE）。使用 QBE 的前提条件是应用程序支持具有某种属性值集合（非默认值）的查询类实例。查询返回所有的匹配属性值的持久实例。QBE 不是特别强大的方法，但是对一些应用程序却很方便。下面的代码片断演示 Hibernate 的 QBE：

```
User exampleUser = new User();
exampleUser.setFirstname("Max");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(exampleUser) );
List result = criteria.list();
```

QBE 的典型用例是允许用户指定属性值范围的查找屏幕，指定属性值范围用来匹配返回的结果集。这种功能在查询语言中很难清晰地表达，操纵字符串需要指定动态的约束集。

QBC API 和这种查询机制的例子将在第七章详细讨论。

现在你知道 Hibernate 中基本的检索选项。我们在本节的剩余部分关注对象图的抓取策略。抓取策略定义了用查询或装载操作检索对象图（或子图）的哪一部分。

4.4.5 抓取策略

传统的关系数据访问利用内连接和外连接检索相关的实体，用单个 SQL 查询抓取对某个计算所需要的所有数据。一些原始的 ORM 实现分开抓取数据，多次请求小块的数据，应用程序作为响应也会多次导航持久对象图。这种方法不能有效利用关系数据库的连接能力。实际上，这种数据访问策略将来很难扩展。ORM 中的一个最困难的问题—可能是最困难的—是提供对关系数据库的有效访问，鉴于应用程序喜欢把数据当成对象图看待。

对于我们经常开发的多种应用程序（多用户，分布式，web 和企业应用），检索对象时多次往返于数据库是不可取的。因此，我们讨论的工具比传统的工具更强调 ORM 中的 R（关系）。

有效地抓取对象图的问题已经通过在关联映射的元数据中指定关联级抓取策略解决了。这种方法存在的问题是每段代码使用一个需要不同集合的相关对象的实体。但是这是不够的。

我们需要的是支持细粒度的运行时关联抓取策略。Hibernate 两者都支持，允许在映射文件中指定默认的抓取策略，然后在代码运行时重载。

Hibernate 对于任何关联允许在四种抓取策略中选择，在关联元数据和运行时：

- 立即抓取—立即抓取关联的对象，使用连续的数据库读（或缓存查找）。
- 延迟抓取—当第一次访问时，“延迟”抓取相关的对象或集合。这个结果在对数据库的新请求中（除非缓存了相关的对象）。
- 提前抓取—相关的对象或集合同拥有它们的对象一起抓取，使用 SQL 外连接，不需要额外的数据库请求。
- 批量抓取—在访问延迟关联时，这种方法通过检索一批对象或集合来提高延迟抓取的性能。（批量抓取也用来提高立即抓取的性能。）

让我们仔细看看每种抓取策略。

立即抓取

立即的关联抓取发生在从数据库中检索实体然后立即在下一个对数据库或缓存的请求中检索另一个（或一些）相关的实体的时候。立即抓取通常不是有效的抓取策略除非希望关联的实体一直被缓存。

延迟抓取

当客户请求数据库中的实体及其相关的对象图时，通常不必检索每个（非直接的）关联对象的整个对象图。你不希望立即把整个数据库装载到内存中，例如，装载单个 Category 不应该触发装载这个目录的所有 Item。

延迟抓取能够让你决定第一次访问数据库时装载多少对象图，并且与其关联的对象只有在第一次访问时才装载。延迟抓取是对象持久化中的基本内容，而且是取得可接受性能的第一步。

我们推荐在开始的时候把映射文档中所有的关联映射为延迟（或可能是批量延迟）抓取。这种策略然后被强制提前抓取发生的查询重载。

提前（外连接）抓取

延迟关联抓取能够帮助减少数据库装载，而且通常是一种好的默认策略。然而，这在性能优化发生前有点盲目猜测。

提前抓取让你显式地指定哪些关联的对象应该同引用它们的对象一起装载。Hibernate 然后在单个数据库请求中使用 SQL 的 OUTER JOIN 返回关联的对象。Hibernate 的性能优化通常包括针对某些事务明智地使用提前抓取。因此，即使在映射文件中声明了默认的抓取策略，在运行时对于某个 HQL 或条件查询指定使用这种策略更普遍。

批量抓取

批量抓取不是严格的关联抓取策略，它是帮助提高延迟（或立即）抓取性能的一种技术。通常，当装载对象或集合的时候，SQL 的 WHERE 子句指定对象的标识符或拥有集合的对象。如果开启了批量抓取，Hibernate 看起来知道什么会在当前 session 中引用其它代理实例或未初始化的集合，尽量通过在 WHERE 子句中指定多个标识符值来同时装载这些对象。

我们不是这种方法的热心者，提前抓取几乎一直是更快的。批量抓取对那些希望用 Hibernate 达到可接受的性能而不用想太多关于要执行的 SQL 的经验不足的用户很有用。（注意，你可能很熟悉批量抓取，因为它已经被许多 EJB2 引擎使用）。

现在我们在映射元数据中对一些关联声明抓取策略。

4.4.6 在映射中选择一种抓取策略

Hibernate 允许你在映射元数据中通过指定属性选择默认的关联抓取策略。可以使用 Hibernate 查询方法的特性重载默认的策略，你将会在第七章看到。一点小警告：不比立即明白本节出现的每个参数，我们推荐你先浏览一下，当你在应用程序中优化默认的抓取策略时把本节当作参考。Hibernate 映射形式的一个缺点就是集合映射函数有点同单点关联不同，因此，我们将要分别覆盖两种情况。让我们首先考虑 Bid 和 Item 之间双向关联的两端。

单点关联

对于<many-to-one>或<one-to-one>关联，如果关联的类映射开启代理延迟抓取是可能的。

对于 Item 类，我们通过指定 lazy=”true”开启代理：

```
<class name="Item" lazy="true">
```

现在记得从 Bid 关联到 Item：

```
<many-to-one name="item" class="Item">
```

当我们从数据库中检索 Bid 时，关联属性可能持有 Hibernate 生成的 Item 子类的实例，这个实例代理所有对从数据库中延迟抓取的不同 Item 实例的方法调用（这是 Hibernate 代理更详细的定义）。

Hibernate 使用两种不同的实例以便能够代理更多态的关联—当代理的对象被抓取时，它可能是 Item 映射子类的一个实例（那就是，如果 Item 有子类的话）。我们甚至可以选择被 Item 类实现的任何接口作为代理的类型。要这么做，使用 proxy 属性声明它，而不是指定 lazy=”true”：

```
<class name="Item" proxy="ItemInterface">
```

只要我们在 Item 中声明了 proxy 或 lazy 属性，任何对 Item 的单点关联都被代理和延迟抓取，除非关联通过声明 outer-join 属性重载了抓取策略。

对于 outer-join 有三种可能值：

- outer-join=”auto”—默认值。当没有指定这个属性时，如果开启了代理 Hibernate 延迟抓取关联的对象，或者如果禁止了代理（默认值）提前使用外连接。
- outer-join=”true”—Hibernate 一直使用外连接提前抓取关联，即使开启了代理。这允许你对相同的代理类的不同关联选择不同的抓取策略。
- outer-join=”false”—Hibernate 从不用外连接抓取关联，即使开启了代理。这对于希望相关的对象存在于第二级缓存中很有用（见第五章）。如果在第二级缓存中不可用，使用额外的 SQL SELECT 立即抓取对象。

因此，如果我们希望再一次开启关联的提前抓取，既然开启了代理，我们就要指定

```
<many-to-one name="item" class="Item" outer-join="true">
```

对于一对一关联（在第六章详细讨论），延迟抓取仅仅当关联的对象已经存在时是理论上

可行的。我们通过指定 constrained=”true”来标识。例如，如果 item 仅仅有一个 bid，Bid 的映射应该为：

```
<one-to-one name="item" class="Item" constrained="true">
```

constrained 属性跟<many-to-one>映射的 not-null 属性有点相似。它告诉 Hibernate 关联的对象是必需的，不能为 null。

为了开启批量抓取，在 Item 的映射中指定 batch-size：

```
<class name="Item" lazy="true" batch-size="9">
```

批量大小限制在单个批量中可能检索的 item 数。这里选择一个合理的小的整数。

当我们考虑集合时可能会遇到相同的属性（outer-join，batch-size 和 lazy），但是解释有点不同。

集合

对于集合来说，抓起策略不仅支持实体关联，而且也支持集合值（例如，字符串的集合可以被外连接抓取）。

就像类一样，集合有它们自己的代理，通常叫做*集合包装*。不像类，集合包装一直是那样，即使延迟抓取关闭了（Hibernate 需要包装来检测集合变化）。

集合映射可以既不声明 lazy 属性又不声明 outer-join 属性或两者都声明（两者都指定没有什么意义）。有意义的参数如下：

- 两个属性都不指定—这个参数跟 outer-join=”false” lazy=”false”等价。集合从第二级缓存中抓取或通过立即的额外的 SQL SELECT 抓取。这个参数是默认的并且当为这个集合开启第二级缓存时最有用处。
- outer-join=”true”—Hibernate 使用外连接提前抓取关联。在写这本书的时候，Hibernate 对于每个 SQL SELECT 仅仅能抓取一个集合，因此用 outer-join=”true”不可能声明属于相同持久类的多个集合。
- lazy=”true”—Hibernate 当第一次访问集合时延迟抓取集合。

我们不推荐对集合使用提前抓取，因此用 lazy=”true”映射 bid 的 item 集合。这个参数几乎一直用于集合映射（应该是默认值，我们推荐你在所有的集合映射中把它当作默认值）：

```
<set name="bids" lazy="true">
<key column="ITEM_ID"/>
<one-to-many class="Bid"/>
</set>
```

我们甚至能够为集合开启批量抓取。这种情况下，批量大小不再指批量抓取中的 bid 数，而是指 bid 的集合数：

```
<set name="bids" lazy="true" batch-size="9">
<key column="ITEM_ID"/>
<one-to-many class="Bid"/>
</set>
```

这个映射告诉 Hibernate 在一次批量抓取中装载九个 bid 集合，取决于多少未初始化的 bid 集合当前出现在与 session 关联的 item 里。换句话说，如果在一个 Session 中有五个具有持久状态的 Item 实例，并且所有的都有未初始化的 bid 集合，如果访问了其中的一个，Hibernate 将在单个 SQL 查询中自动装载所有的五个集合。如果有十一个 item，仅仅会抓取九个。批量抓取会显著地减少请求层次对象的查询数（例如，当装载父子 Category 对象树时）。

让我们讨论一种特殊情况：多对多关联（我们在第六章详细讨论这种映射）。通常使用连接表（一些开发者也把它叫做关系表或关联表）保存两个关联表的键值并且因此允许多对多的多样性。如果你决定使用提前抓取就不得不考虑这个附加表。看看下面这个简单的多对多示例，映射从 Category 到 Item 的关联：

```
<set name="items" outer-join="true" table="CATEGORY_ITEM">
<key column="CATEGORY_ID"/>
<many-to-many column="ITEM_ID" class="Item"/>
</set>
```

在这个例子中，提前抓取策略仅仅引用关联的表 CATEGORY_ITEM。如果我们用这种抓取策略装载 Category，Hibernate 将会用一个外连接 SQL 查询自动抓取所有来自

CATEGORY_ITEM 的连接入口，而不是来自 ITEM 的 item 实例！

多对多关联中包含的实体当然也能用相同的 SQL 查询提前抓取。<many-to-many>元素允许自定义这种行为：

```
<set name="items" outer-join="true" table="CATEGORY_ITEM">
<key column="CATEGORY_ID"/>
<many-to-many column="ITEM_ID" outer-join="true" class="Item"/>
</set>
```

现在当装载 Category 时 Hibernate 用单个外连接查询抓取所有 Category 中的 Item。然而，记住我们通常推荐把延迟加载作为默认的抓取策略并且 Hibernate 限制每个映射持久类只能有一个提前抓取的集合。

设置抓取深度

现在我们讨论全局抓取策略设置：最大抓取深度。这个设置控制 Hibernate 在单个 SQL 查询中使用的外连接表数。考虑从 Category 到 Item 和从 Item 到 Bid 的整个关联链。第一个是多对多关联，第二个是一对多关联，因此两种关联都要映射成集合元素。如果我们为两种关联都声明 outer-join=”true”（不要忘记指定<many-to-many>声明）并装载单个 Category，Hibernate 将会执行多少查询？仅仅提前抓取 Item，或者也提前抓取所有 Bid 中的每个 Item？

你可能希望使用包含 CATEGORY，CATEGORY_ITEM，ITEM 和 BID 表的外连接操作的单个查询。然而，这不是默认的情况。

Hibernate 的外连接抓取行为受全局配置参数 hibernate.max_fetch_depth 的控制。如果把它设为 1（也是默认值），Hibernate 就只抓取 Category 和来自 CATEGORY_ITEM 关联表的连接入口。如果把它设为 2，Hibernate 用相同的 SQL 查询执行也包含 Item 的外连接。把这个参数设为 3，在一个 SQL 语句中连接所有的四个表并且装载所有的 Bid。

抓取深度的推荐值依赖于连接性能和数据库表的大小，先用低的值（小于 4）测试应用程序，然后当调整你的应用程序时增加或减少这个值。全局最大抓取深度也可以用于使用提前抓取策略映射的单端关联（many-to-one,<one-to-one>）。

记住在映射元数据中声明的提前抓取策略只有当使用标识符检索、使用条件查询 API 或手动导航对象图时才有效。任何 HQL 查询可以在运行时指定自己的查询策略，这样忽略映射的默认值。

初始化延迟关联

4.4.7 调整检索的对象

让我们看看在应用程序中调整检索的对象需要的几步操作：

1. 像第二章描述的那样开启 Hibernate SQL 日志功能。你也应该准备阅读、理解和评估 SQL 查询及对于特定的关系模型的执行性能：单个的连接操作会比两个选择快么？在所有的索引正确使用的情况下，数据库的缓存命中率将会怎样？让 DBA 帮你评估性能，她只具有能够决定执行哪个 SQL 是最好的知识。
2. 单步跟踪你的应用用例，注意 Hibernate 执行了多少条 SQL 语句及执行了什么样的 SQL 语句。用例可能是 web 应用中的一个页面也可能是一系列用户对话框。这步也包括收集用例中使用的检索对象的方法：遍历图形，通过标识符、HQL 及条件查询检索。你的目标是通过调整元数据中的默认抓取策略降低 SQL 查询数及复杂度。
3. 你可能会遇到下面两个常见的问题：
   - 如果使用连接操作的 SQL 语句太复杂太慢，把<many-to-many>关系的 outer-join 设为 false（默认情况是开启的）。尽量调整全局 hibernate.max_fetch_depth 配置选项，记住最好把值设在 1 和 4 之间。
   - 如果执行太多的 SQL 语句，对所有的集合映射使用 lazy=”true”。Hibernate 对集合元素默认使用立即的附加抓取（那就是，如果它们是实体，可以级联到图形中）。很少情况下，如果你能确信，设置 outer-join=”true”并关闭对特定集合的延迟加载。记住每个持久类只有一个集合属性可能被明确抓取。如果给定的工作单元包括几个“相同的”集合或者访问父子对象树，使用值在 3 和 10 之间的批量抓取来进一步优化集合抓取。

4. 设置完新的抓取策略，返回用例，再一次检查生成的 SQL。注意 SQL 语句，走到下一个用例。
5. 优化完所有的用例，再检查一遍看是否某个优化对其它优化有影响。对于同样的经历，你就能够避免负面影响并及时纠正。

这种优化技术不仅对默认的抓取策略是可行的，也可以用它来调整 HQL 和条件查询，对特定的用例和工作单元忽略及覆盖默认的抓取。我们将在第七章讨论运行时抓取。

本节，我们已经开始考虑性能问题，尤其是与关联抓取相关的问题。当然，抓取对象图最好的方式是从内存缓存中抓取，如下一章所示的那样。

4.5 总结

对象/关系不匹配的动态问题跟更加熟悉和理解结构化不匹配问题一样重要。本章我们主要关心与持久机制相关的对象生命周期。现在你理解了三种 Hibernate 定义的对象状态：持久的，分离的和瞬时的。当你调用 Session 接口的方法或创建和删除对已经持久化的实例的引用时，对象在不同状态之间转换。后面的行为由可配置的级联形式控制，Hibernate 的传递持久化模型。这个模型允许声明以关联为基础的级联操作（像保存或删除），比传统的可到达性持久化模型更强大更灵活。你的目标是为每个关联找到最好的级联形式，因而最小化存储对象时不得不调用持久管理器的次数。

从数据库中检索对象是同样重要的：可以通过访问属性来遍历业务模型图，让 Hibernate 显式地抓取对象。也可以通过标识符装载对象，用 HQL 写任意的查询或者使用条件查询 API 创建面向对象的表示层。除此之外，对于特别的情况可以使用本地 SQL 查询。

大多数哲学对象检索方法使用映射远射击中定义的默认抓取策略（HQL 忽略它们，条件查询能够重载它们）。正确的抓取策略最小化通过延迟、提前或批量抓取对象不得不执行的 SQL 语句数量。你可以通过分析每个用例中执行的 SQL，调整默认的和运行时抓取策略来优化 Hibernate 应用程序。

下面我们讨论与事务和缓存相关的话题。